Formal techniques for Java-like programs

Report**Author(s):**

Eisenbach, Susan; Leavens, Gary T.; Müller, Peter; Poetzsch, Michael; Poll, Erik

Publication date:

2003

Permanent link:

<https://doi.org/10.3929/ethz-a-006666086>

Rights / license:

In Copyright - Non-Commercial Use Permitted

Originally published in:

Technical Report / ETH Zurich, Department of Computer Science 408

Formal Techniques for Java-like Programs 2003

Technical Report 408
Department of Computer Science

Susan Eisenbach, Gary T. Leavens, Peter Müller, Arnd Poetzsch-Heffter,
and Erik Poll (editors)

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Formal Techniques for Java-like Programs 2003

Proceedings, Darmstadt, Germany
July 21, 2003

Susan Eisenbach, Gary T. Leavens, Peter Müller, Arnd Poetzsch-Heffter,
and Erik Poll (editors)

Copyright © 2003 each paper's copyright is held by its authors.

Session 1: Language Semantics

Inner Classes visit Aliasing

Matthew Smith and Sophia Drossopoulou

Java Definite Assignment in Isabelle/HOL

Norbert Schirmer

Flexible, source level dynamic linking and re-linking

Sophia Drossopoulou

Algebraic Semantics of the Statements of Sequential Java

Kazem Lellahi, Alexandre Zamulin

Session 2: Specification and Verification

JML Support for Primitive Arbitrary Precision Numeric Types: Definition and Semantics

Patrice Chalin

Verifying JML specifications with model fields

Cees-Bart Breunesse, Erik Poll

Verification of Object-Oriented Programs with Invariants

Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte

A Tool-supported Assertion Proof System for Multithreaded Java

Erika Abraham, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen

Session 3: Refinement and Static Analysis

Toward Automatic Generation of Provably Correct Java Card Applets

Alessandro Coglio

Ownership: transfer, sharing, and encapsulation

Anindya Banerjee and David A. Naumann

Static Detection of Atomicity Violations in Object-Oriented Programs

Christoph von Praun and Thomas R. Gross

Checking Concise Specifications for Multithreaded Software

Stephen Freund and Shaz Qadeer

Session 4: Language Implementation and Runtime Checking

Stronger Typings for Separate Compilation of Java-like Languages

Davide Ancona and Giovanni Lagorio

Static analysis for eager stack inspection

Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari

Abstraction-carrying Code: A New Method to Certify Temporal Properties

Songtao Xia, James Hook

Instrumentation of Java Bytecode for Runtime Analysis

Allen Goldberg and Klaus Havelund

Inner Classes visit Aliasing

Matthew Smith, Sophia Drossopoulou

Imperial College of Science Technology and Medicine

Abstract. Inner classes appear nested within class definitions. They may access any members of the classes in which they are contained. The interplay between inner classes, aliasing and subclasses can make resolution of such accesses intricate.

We offer a succinct model for member classes, in which we highlight these intricacies and we prove soundness.

1 Introduction

Inner classes were introduced into Java with version 1.1, and were motivated mainly by the AWT and later Swing[1]. Java's event model requires the use of many small listener and adaptor classes for which inner classes are perfectly suited. This continues to be the way inner classes are most often used. These applications usually only apply the basic features of inner classes. Thus, many developers may be surprised by the issues arising from inner classes. Inner classes (or their equivalents) also exist in other object-oriented languages, notably Beta [2]. They can be useful and powerful.

An inner class in Java can access any of the members of any class in which it is contained¹. To allow this, an instance of an inner class has to be bound to (have a host) an instance of the containing class. Through inheritance an object may require *several* hosts; these may be of the same class. Moreover, through inheritance, an object may have *one* object fulfilling the role of *many* hosts. Such issues cannot be reflected in a functional model of inner classes, e.g. [3]. They are reflected in the model presented here.

In Section 2 we present the basics as well as some more complex behaviours of inner classes, and show examples. In Section 3 we discuss $\mathcal{L}_{jm}$ design decisions. Section 4 presents $\mathcal{L}_{jm}$, a compact object oriented language with inner classes and aliasing. We present the operational semantics and type system with particular focus on inner classes. In Section 5 we state important properties of $\mathcal{L}_{jm}$, and esp. soundness. In section 6 we describe translation to top-level classes. In Section 7 we offer observations we have made about inner classes. In section 8 we draw conclusions and outline further work.

A version of this paper including the Appendices is available at [http : //www.doc.ic.ac.uk/ mjs198/InnerClasses](http://www.doc.ic.ac.uk/~mjs198/InnerClasses).

2 Introduction to Java Inner Classes ²

2.1 Basic Concepts

A class, $A.B$, is *inner* to a class A if it is declared inside class A ; class A is then called an *enclosing class* of $A.B$. An instance of an inner class has a link to an instance of the enclosing class which we call the *host* object. Thus, an object of class $A.B$ has a host of class A , which it can access through the expression $A.this$. We say that $A.B$ *requires* a host of class A .

Inner classes may extend and may be extended by other classes. A class extending an inner class 'inherits' the requirement for host objects. These inherited hosts ensure that inherited methods which access the (inherited) host can be executed. Methods defined in the subclass may not access

¹ particularly useful for AWT/Swing applications as it avoids explicit passing of call back objects

² Java supports four kinds of inner class: member, local, anonymous and static member classes. In this work, as in FJI, we consider member classes only.

the inherited host since the subclass is not inner to (not lexically contained in) the inherited enclosing class.

In Fig. 1³ classes `M.N` and `M.O` are inner classes of `M` (and as such have names of the form `<enclosing class name>.<simple name>`) and `M.O` extends `M.N`. This is shown diagrammatically in a UML-like style on the RHS of the figure. Objects of class `M.O` require *two* host objects of class `M`: one because `M.O` extends `M.N`, and one because `M.O` is inner to `M`.

The methods `f` and `g` access host objects: In `f` the expression `M.this` returns the host object of type `M` required by class `M.N`, while in `g` the expression `M.this` returns the host object of type `M` required by class `M.O`. Also, in `f` the expression `M.N.this` is a synonym for `this` – view a class as trivially inner to itself.

```
class M {
  int i;
  class M.N {
    int f() {
      return M.this.i;
    }
  }
  class M.O extends M.N {
    int g() {
      return M.this.i;
    }
  }
}
```

Fig. 1. Code and Diagram for Interesting Inner Classes

Figure 2 shows a possible object configuration, in a UML-like style: 1 is an object of class `M`, with field `i` containing the value 1. The arrows indicate links to hosts. Some objects have two hosts, e.g. 4 has 1 and 2 as hosts. In order to distinguish between the host required by the object's class and the inherited hosts, we annotate the links with the name of the class requiring the host. For example, for object 4, the `M.N` host is object 1 and the `M.O` host (inherited) is object 2.⁴

Table 3 shows the results of calling methods `f` and `g` on these objects. Notice, that although the method body for `f` and `g` are identical, the call of `f` on 4 returns a different result than the call of `g` on 4. This is so, because `M.this` in `f` indicates the host required by `M.N`, whereas `M.this` in `g` indicates the host required by class `M.O`. Therefore, evaluation of access to hosts depends on which class defined the method being executed.

Fig. 2. Possible object configurations for Fig. 1

Receiver	Method	Result
3	f	1
4	f	1
5	f	3
4	g	3
5	g	3

Fig. 3. Method call results

³ In Java qualified names, e.g. `M.N`, are optional in some contexts and illegal in class declarations. We use qualified names here for consistency with Sec. 4. We purposefully omit constructors.

⁴ Annotating the link with the class of the hosts (rather than with the class requiring the host) would not work because several hosts may have the same class, as for example the hosts of class `M.O`.

2.2 A Note on Terminology

The terminology of this paper is a little different to that of the JLS [4]. The JLS terminology is more descriptive but consequently more complex. The JLS talks of *enclosing instances* rather than hosts. In order to accurately describe the semantics of the language the JLS further qualifies terminology, for example the *nth lexically enclosing instance* is the host relating to the class in which the current class is nested n levels deep. We choose to describe demonstrate the semantics informally through example and formally through operational semantics and type rules.

2.3 Advanced Concepts: Aliasing and Deep Nesting of Inner Classes

The interplay of deep inner classes (those which are nested within other inner classes) with aliasing cause some more complex properties, which we demonstrate through the classes from Fig. 4 and objects in Fig. 5. The properties are:

- P1** Inner classes may be nested to any level.
Here, A.B.C.D is inner to A.B.C which is inner to A.B while A.B is inner to A.
- P2** An object may have more than one host.
Here, objects 2 and 4 have two hosts, object 7 has three hosts.
- P3** The host of an object may be of a subclass of the corresponding enclosing class.
Here, the host of 6 is 5, which has class F, a subclass of A.B.C.
- P4** An object may access through the host mechanism several - but not all - objects in the transitive closure of the host relation. Namely, an object of class $A_1 \dots A_n$ may access any host $A_1 \dots A_k$.this for $k \leq n$; also, if $A_1 \dots A_n$ is a subclass of $B_1 \dots B_m$, when executing an inherited method it may access any host $B_1 \dots B_j$.this.
In our example, the methods of A.B.C.D can access host objects A.this, A.B.this and A.B.C.this. For object 6 these hosts are 1, 4 and 5 respectively. A method of A.B.C inherited in class F can access hosts A.this and A.B.this, so, object 5 could reach objects 1 and 4 through the host mechanism.
- P5** All other objects in the transitive closure of the host relationship cannot be accessed through the host mechanism.
Here, because F is not inner to A.B, methods in F cannot contain the expression A.B.this. Therefore, object 5 will never access object 3 through the host mechanism.

3 Main Design Decisions for $\mathcal{L}_{jm}$

The issues discussed in the previous section motivated our design choices for our formalism, $\mathcal{L}_{jm}$.

Object Representation An object may have many hosts which can be distinguished by the name of the class requiring them. Consequently we represent objects in the form:

$$\llbracket C \mid C_1 : \iota_1, \dots, C_n : \iota_n \mid f_1 : v_1, \dots, f_r : v_r \rrbracket$$

where C is the class of the object. $C_1 : \iota_1, \dots, C_n : \iota_n$ are the hosts required by $C_1, \dots, C_n$ respectively, and $f_1 : v_1, \dots, f_r : v_r$ are the names and values of fields.

The Store Stores capture both heaps and stacks:

$$\begin{aligned} store &= (\{ \text{this} \} \rightarrow addr) \cup (\{ x \} \rightarrow val) \cup (addr \rightarrow object) \\ val &= \{ null \} \cup addr \\ dev &= \{ nullPtrExc, stuckErr \} \\ object &= \{ \llbracket C \mid C_1 : \iota_1, \dots, C_k : \iota_k \mid f_1 : v_1, \dots, f_r : v_r \rrbracket \} \\ addr &= \mathbb{N} \end{aligned}$$

where $v, v', v_1, \dots$ range over val , $\iota, \iota', \iota_1, \dots$ range over $addr$, $\sigma, \sigma', \sigma_1, \dots$ range over $store$ and $C, C', C_1, \dots$ range

Fig. 4. A complex inner class hierarchy

Fig. 5. Possible object configuration

over class names and $f, f', f_1, \dots$ range over field names. The identifier this refers to the current receiver and x to the current method argument. Store lookup, $\sigma(\iota)(f)$, and update, $\sigma[\iota \mapsto \sigma(\iota)[f \mapsto v]]$, have the obvious meaning.

Assuming no fields in the classes, the objects of Fig. 5 would be represented in σ_0 as:

$$\begin{aligned}
 \sigma_0(1) &= \llbracket A \rrbracket \\
 \sigma_0(2) &= \llbracket A.E \mid A.E : 1, A.B : 1 \rrbracket \\
 \sigma_0(3) &= \llbracket A \rrbracket \\
 \sigma_0(4) &= \llbracket A.E \mid A.E : 3, A.B : 1 \rrbracket \\
 \sigma_0(5) &= \llbracket F \mid A.B.C : 4 \rrbracket \\
 \sigma_0(6) &= \llbracket A.B.C.D \mid A.B.C.D : 5 \rrbracket \\
 \sigma_0(7) &= \llbracket F.G \mid F.G : 5, A.E : 1, A.B : 3 \rrbracket
 \end{aligned}$$

Operational Semantics Format Execution needs to keep track of the class defining the method currently being executed. Therefore, the operational semantics has the format:

$$P, C \vdash e, \sigma \rightsquigarrow r, \sigma'$$

where P is the program, C is the class defining the method currently being executed, e is the expression to be evaluated, σ is the store before evaluation, r is the result of evaluation (a value or a deviation) and σ' the store after evaluation.

Host Object Lookup Host lookup depends on the store, the address ι whose host is looked up, the class C containing the expression being executed, and the class C' of the host being looked up. For store σ as above, we have: The function is defined as follows;

$$\mathcal{H}(\sigma, \iota, C, C') = \begin{cases} \iota & \text{if } C = C' \\ \mathcal{H}(\sigma, \iota', C'', C') & \text{if } C \neq C' \wedge C = C''.c \\ & \wedge \sigma(\iota) = \llbracket C''' \mid \dots C : \iota' \dots \rrbracket \\ \text{udf} & \text{otherwise} \end{cases}$$

In our example:

$$\begin{aligned}
 \mathcal{H}(\sigma_0, 6, \text{A.B.C.D}, \text{A}) &= \mathcal{H}(\sigma_0, 5, \text{A.B.C}, \text{A}) \\
 &= \mathcal{H}(\sigma_0, 4, \text{A.B}, \text{A}) \\
 &= \mathcal{H}(\sigma_0, 1, \text{A}, \text{A}) \\
 &= 1
 \end{aligned}$$

4 $\mathcal{L}_{jm}$, our formal model of inner classes

In this section we present $\mathcal{L}_{jm}$, a compact, class-based, object-oriented language with aliasing and member classes. $\mathcal{L}_{jm}$ is an extension of earlier work on the formalization of Java subsets, e.g. [5, 6]. $\mathcal{L}_{jm}$ is similar to Java. We ignore features of Java which are not immediately relevant. e.g. threads, casting etc.

For the sake of simplicity, we impose the following restrictions 1-8. Also, the deviations 9-12 differ slightly from Java, and allow for a simpler model:

1. The receiver of field access and method calls must be made explicit (no implicit `this`).
2. In instantiation of inner classes, the host object must be made explicit (no implicit `this`, we require `this.new A.B()`).
3. Full names must be used for inner class types in field and method declarations.
4. Fully qualified names must be used for all host object accesses e.g. always `A.B.this`, even when Java would allow `B.this` as a synonym.
5. No unqualified `this` expression, `<class name>.this` is a synonym.
6. No sequences of expressions.
7. No constructors - objects are instantiated with default values.
8. Methods have precisely one parameter.
9. Methods implicitly return the result of evaluating their body.
10. Inner classes are declared with their full name, e.g. `A.B` as opposed to just `B` in class `A`.
11. Inner classes are instantiated with their full name, so `a.new A.B()` rather than `a.new B()`.
12. Inherited hosts are provided to `new` through a sequence of expressions.

4.1 $\mathcal{L}_{jm}$ Syntax

```

progr ::= class*
class ::= class C extends C { field* meth* class* }
field ::= C f
meth ::= C m (Cx){e}
C      ::= c | c.C
e      ::= e.f | e.f := e | e.m(e) | new c(e*) | e.new C(e*) | x | C.this | null

```

The expression `new c(e1, ..., en)` instantiates non-inner classes (hence the restriction to simple class names). The expression `e0.new C(e1, ..., en)` instantiates inner classes, where `e0` is the host object for `C`, as in Java, and `e1, ..., en` supply inherited host objects - if any. The inherited hosts are supplied starting from the most specific superclass. For example, instantiation of class `F.G` from Fig. 4 would have the form `< F.G host > .new F.G(< A.E host >, < A.B host >)`.

4.2 Lookup Functions and Judgements on Programs

Full definitions of the following judgements are available in [7]

The judgement $P \vdash C \diamond_c$ states that `C` is a valid class name in the program. The judgement $P \vdash C \leq C'$ expresses that `C` is a subclass of `C'` and the relation is reflexive and transitive. Similarly, $P \vdash C \preceq C'$ expresses that `C` is inner to `C'` and again the relation is reflexive and transitive. The judgement $\vdash P \diamond_n$ expresses that all class names are unique, that field and method names are unique

per class, that the name of an inner class is the name of the class that contains it extended with a simple class name and that the inheritance hierarchy is acyclic. We implicitly require $\vdash P \diamond_n$.

The function $\mathcal{C}(P, C)$ returns the definition of class C in program P . The function $\mathcal{MD}(P, C, m)$ returns the definition of the method m in the class C of program P . The function $\mathcal{M}(P, C, m)$ returns the pair consisting of the name of the class containing the method definition and the definition of the method. The function $\mathcal{F}(P, C, f)$ returns the type of field f in class C . $\mathcal{Fs}(P, C)$ returns the set of all fields names declared in C and its superclasses. All above classes return *udf* if C is not defined in P .

$\mathcal{SIs}$ produces the list of the super classes of a given class which are also inner classes, ordered on the inheritance relation. In other words, $\mathcal{SIs}$ is the list of superclasses requiring hosts.

$$\begin{aligned} \mathcal{S}(P, C) &= \begin{cases} C' & \text{if } \mathcal{C}(P, C) = \text{class } C \text{ extends } C' \dots \\ \text{udf} & \text{if } \mathcal{C}(P, C) = \text{udf} \text{ or } C = \text{Object} \end{cases} \\ \mathcal{SIs}(P, C) &= \begin{cases} C :: \mathcal{SIs}(P, \mathcal{S}(P, C)) & \text{if } P \vdash C \diamond_c \text{ and } C = C'.c \\ \mathcal{SIs}(P, \mathcal{S}(P, C)) & \text{if } P \vdash C \diamond_c \text{ and } C = c \\ \epsilon & \text{otherwise} \end{cases} \end{aligned}$$

4.3 $\mathcal{L}_{jm}$ Operational Semantics

$$\frac{}{P, C \vdash v, \sigma \rightsquigarrow v, \sigma} \text{val} \quad \frac{}{P, C \vdash x, \sigma \rightsquigarrow \sigma(x), \sigma} \text{var} \quad \frac{\mathcal{H}(\sigma, \sigma(\text{this}), C, D) = \iota}{P, C \vdash D.\text{this}, \sigma \rightsquigarrow \iota, \sigma} qThis$$

$$\frac{P, C \vdash e, \sigma \rightsquigarrow \iota, \sigma' \quad \sigma'(\iota)(f) \neq \text{udf}}{P, C \vdash e.f, \sigma \rightsquigarrow \sigma'(\iota)(f), \sigma'} fld \quad \frac{P, C \vdash e, \sigma \rightsquigarrow \iota, \sigma'' \quad P, C \vdash e', \sigma'' \rightsquigarrow v, \sigma''' \quad \sigma'''(\iota)(f) \neq \text{udf} \quad \sigma' = \sigma'''[\iota \mapsto \sigma'''(\iota)[f \mapsto v]]}{P, C \vdash e.f := e', \sigma \rightsquigarrow v, \sigma'} fldass$$

$$\frac{\begin{array}{l} P, C \vdash e, \sigma \rightsquigarrow \iota, \sigma_0 \\ P, C \vdash e', \sigma_0 \rightsquigarrow v', \sigma_1 \\ \sigma_1(\iota) = \llbracket C' \parallel \dots \parallel \dots \rrbracket \\ \mathcal{M}(P, C', m) = (t \ m(t_1 \ x) \ \{e''\}, C'') \\ \sigma' = \sigma_1[\text{this} \mapsto \iota][x \mapsto v'] \\ P, C'' \vdash e'', \sigma' \rightsquigarrow v, \sigma'' \end{array}}{P, C \vdash e.m(e'), \sigma \rightsquigarrow v, \sigma''[\text{this} \mapsto \sigma(\text{this})][x \mapsto \sigma(x)]} methcall$$

$$\frac{\begin{array}{l} P \vdash c \diamond_c \\ P, C \vdash e_i, \sigma_{i-1} \rightsquigarrow \iota_i, \sigma_i \ 1 \leq i \leq n \\ \mathcal{SIs}(P, c) = C_1 :: \dots :: C_n \\ \mathcal{Fs}(P, c) = \{f_1, \dots, f_r\} \\ v = \text{null}, \ \iota \text{ new in } \sigma_n \\ o = \llbracket c \parallel C_1 : \iota_1, \dots, C_n : \iota_n \parallel f_1 : v, \dots, f_r : v \rrbracket \end{array}}{P, C \vdash \text{new } c(e_1, \dots, e_n), \sigma_0 \rightsquigarrow \iota, \sigma_n[\iota \mapsto o]} newT$$

$$\frac{\begin{array}{l} P \vdash C' \diamond_c \\ P, C \vdash e_0, \sigma \rightsquigarrow \iota_0, \sigma_0 \\ P, C \vdash e_i, \sigma_{i-1} \rightsquigarrow \iota_i, \sigma_i \ 1 \leq i \leq n \\ \mathcal{SIs}(P, C') = C_0 :: \dots :: C_n \\ \mathcal{Fs}(P, C') = \{f_1, \dots, f_r\} \\ v = \text{null}, \ \iota \text{ new in } \sigma_n \\ o = \llbracket C' \parallel C_0 : \iota_0, \dots, C_n : \iota_n \parallel f_1 : v, \dots, f_r : v \rrbracket \end{array}}{P, C \vdash e_0.\text{new } C'(e_1, \dots, e_n), \sigma \rightsquigarrow \iota, \sigma_n[\iota \mapsto o]} newI$$

In rule *qThis* the search for a host object starts from the current receiver $\sigma(\text{this})$, with C the class containing the expression $D.\text{this}$, and D the type of the host object. For method calls, *methCall*, the expressions for the receiver and the argument are evaluated and used to construct a context for the evaluation of the method body. Note that in the rewrite for the method body, the method environment is updated to C'' , the class returned by $\mathcal{M}$. For object creation, *newI* and *newT* build suitable objects and place them in a new address in the store. The inherited hosts are the results of evaluating $e_1, \dots, e_n$, while $\mathcal{SI}_S$ finds all superclasses requiring hosts.

4.4 $\mathcal{L}_{jm}$ Types

The only types in $\mathcal{L}_{jm}$ are classes. Since $P \vdash C \diamond_c$ guarantees that C is a class it also guarantees that C is a type. Typing judgements in $\mathcal{L}_{jm}$ take the form:

$$P, \Gamma \vdash e : C$$

where P is the program, Γ is a type environment mapping $\{\text{this}, x\}$ to types, e is the expression being typed and C is the type assigned to the expression.

$$\frac{P \vdash C \diamond_c}{P, \Gamma \vdash \text{null} : C} \text{null}$$

$$\frac{}{P, \Gamma \vdash x : \Gamma(x)} \text{var}$$

$$\frac{P \vdash \Gamma(\text{this}) \preceq C}{P, \Gamma \vdash C.\text{this} : C} \text{qThis}$$

$$\frac{P, \Gamma \vdash e : C' \quad \mathcal{F}(P, C', f) = C}{P, \Gamma \vdash e.f : C} \text{fld}$$

$$\frac{\begin{array}{l} P, \Gamma \vdash e : C \\ P, \Gamma \vdash e' : C' \\ \mathcal{F}(P, C, f) = C'' \\ P \vdash C' \leq C'' \end{array}}{P, \Gamma \vdash e.f := e' : C'} \text{fldAss}$$

$$\frac{\begin{array}{l} P \vdash c \diamond_c \\ \mathcal{SI}_S(P, c) = C_1 :: \dots :: C_n \\ C_i = C'_i.c_i \quad 1 \leq i \leq n \\ P, \Gamma \vdash e_i : C'_i \\ P \vdash C'_i \leq C'_i \quad 1 \leq i \leq n \end{array}}{P, \Gamma \vdash \text{new } c(e_1, \dots, e_n) : c} \text{newT}$$

$$\frac{\begin{array}{l} P \vdash C \diamond_m \\ \mathcal{SI}_S(P, C) = C_0 :: \dots :: C_n \\ C_i = C'_i.c_i \quad 0 \leq i \leq n \\ P, \Gamma \vdash e_i : C'_i \\ P \vdash C'_i \leq C'_i \quad 0 \leq i \leq n \end{array}}{P, \Gamma \vdash e_0.\text{new } C(e_1, \dots, e_n) : C} \text{newI}$$

$$\frac{\begin{array}{l} P, \Gamma \vdash e : C \\ P, \Gamma \vdash e' : C' \\ \mathcal{M}(P, C, m) = (C'' m (C''' x) \{e''\}, \dots) \\ C' \leq C''' \end{array}}{P, \Gamma \vdash e.m(e') : C''} \text{meth}$$

The type rule *qThis* allows qualification of *this*, only if the qualification is a prefix (i.e. an enclosing class) of the type of *this* i.e. $\Gamma(\text{this})$. The type rules for object creation, *newT* and *newI*, check the type of the host object expressions and ensure that the correct number of expressions are supplied, which is the number of class names in the list returned by $\mathcal{SI}_S$.

4.5 Agreement, and well-formedness of program

Judgements pertaining to agreement can be found in [7]: $P, \sigma \vdash \iota \triangleleft C'$, expresses that the object at address ι conforms to class C' i.e. its fields and hosts agree with their expected types. The judgement $P, \Gamma \vdash \sigma \diamond$ expresses that all objects agree with their types.

Judgements pertaining to well-formedness can also be found in [7]: A class is well-formed, $P \vdash C \diamond$, if it satisfies the requirements on naming of classes and members, and if the body of each of its methods has a subtype of the declared return type given an appropriate environment. Additionally, field names must not be used in superclasses (no field shadowing), and should a method name have been defined in a super class, then the argument and return types must be the same (method overriding but not overloading). A program is well formed, $P \diamond$, if all its classes are well-formed.

5 Properties of $\mathcal{L}_{jm}$

In this section we first show determinism of the operational semantics. For this, we define ?? equivalence of stores, $\sigma =_\alpha \sigma'$, up to renaming of addresses, where α is a bijection across addresses, definitions are available in the full version [7]. We then combine progress and soundness of typing into one result.

Determinism of the Operational Semantics. *If*

$P, C \vdash e, \sigma \rightsquigarrow r, \sigma',$ and $P, C \vdash e, \sigma'' \rightsquigarrow r', \sigma''',$ and $\sigma =_\alpha \sigma''$ for some α ,
then

$r =_{\alpha'} r',$ and $\sigma' =_{\alpha'} \sigma''',$ for some α' extending α .

Subject Reduction and Progress. *If*

$\vdash P \diamond, P, \Gamma \vdash e : C,$ and $P, \Gamma \vdash \sigma \diamond,$ and $P, D \vdash e, \sigma \rightsquigarrow r, \sigma'$
then

($r = v$ and $P, \sigma' \vdash v \triangleleft C$ and $P, \Gamma \vdash \sigma' \diamond$) or ($r = \text{nullPtrExc}$ and $P, \Gamma \vdash \sigma' \diamond$)

6 Translation from $\mathcal{L}_{jm}$ to $\mathcal{L}_j$

SUN implements inner classes in Java is via translation to equivalent top level classes [8]. We offer a (nearly) equivalent translation from $\mathcal{L}_{jm}$ to $\mathcal{L}_j$ ($\mathcal{L}_j$ is $\mathcal{L}_{jm}$ without member classes). We follow the rules which Sun uses for conversion of qualified names and other inner class features. To simplify matters we assume sequences of expressions and multi-argument methods in $\mathcal{L}_j$. Additionally, we assume the existence of a method in the Object class which returns the receiver. Our translation is not quite correct due to the simplicity of the language. Our lack of access modification makes the host fields globally accessible, though the use of the protected \$ symbol affords some control at the source level.

In [7] we define the following functions, where $\mathcal{P}, \mathcal{C}, \mathcal{T}, \mathcal{M}$ and $\mathcal{E}$ denote sets of programs, class definitions, types, methods and expressions, and the subscripts denote $\mathcal{L}_{jm}$, and respectively for $\mathcal{L}_j$.

$$\begin{array}{lll} T : \mathcal{P}_{jm} \rightarrow \mathcal{P}_j & T_C : \mathcal{P}_{jm} \times \mathcal{C}_{jm} \rightarrow \mathcal{C}_j & T_T : \mathcal{T}_{jm} \rightarrow \mathcal{T}_j \\ T_M : \mathcal{T}_{jm} \times \mathcal{M}_{jm} \rightarrow \mathcal{M}_j & T_E : \mathcal{T}_{jm} \times \mathcal{E}_{jm} \rightarrow \mathcal{E}_j & \end{array}$$

We can then prove correctness of the translation as follows:

Theorem 1 *If $\vdash P \diamond$ then $\vdash T(P) \diamond$*

Theorem 2 *If*

$\Gamma_{jm}(\text{this}) = C_1, \quad \Gamma_{jm}(x) = C_2, \quad \Gamma_j(\text{this}) = T_T(C_1), \quad \Gamma_j(x) = T_T(C_2), \quad P_{jm}, \Gamma_{jm} \vdash e : C,$
then

$T(P_{jm}), \Gamma_j \vdash T_E(P, e) : T_T(C)$

We now show equivalence of the translation under execution. For this, in Appendix ??, we define equivalent stores with respect to α , a bijection between addresses, and the relation $=_\alpha^T$ which express equivalence of $\mathcal{L}_{jm}$ and $\mathcal{L}_j$ stores.

Theorem 3 *If*

$T(P_{jm}) = P_j, \quad T_E(P_{jm}, e_{jm}) = e_j, \quad \sigma_{jm} =_\alpha^T \sigma_j, \quad P_{jm}, C \vdash e_{jm}, \sigma_{jm} \rightsquigarrow v_{jm}, \sigma_{jm}',$
then

$T(P_{jm}) \vdash e_j \rightsquigarrow v_j, \sigma_j', \quad \text{and there exists } \alpha' \text{ extending } \alpha \text{ such that: } \sigma_{jm}' =_{\alpha'}^T \sigma_j' \quad v_{jm} =_{\alpha'}^T v_j$

7 Observations

In the process of our work on the formalization of inner classes we made the following observations with respect to their design, and the reasons for various restrictions imposed by the language:

- O1** Any class which was part of a cycle in the the union of the subclass and inner class relation, cannot be instantiated. This is so, because instantiation of such a class would require an instance of that class (as a host or inherited host) since this must pre-exist the new instance such a class can never be instantiated by code.
- O2** Although Java forbids classes as in **O1**, a language that allowed them, without allowing cycles in the inheritance hierarchy, would still be sound. Namely, such classes do not interact adversely with other code, they simply cannot be instantiated, and thus are redundant. Should an instance of such a class somehow “magically” exist, execution of well-typed expressions would return values of appropriate classes.⁵
- O3** The host relation in object graphs generated by code forms a DAG. Namely, cycles cannot be generated by code since host objects must pre-exist the objects they are host to. We can easily show that such cycles cannot be created through execution of $\mathcal{L}_{jm}$ expressions.
- O4** However, soundness does not depend on **O3**. Namely, our proof of subject reduction holds even for stores where there are cycles in the host relation.

8 Conclusions and further work

This paper describes the basic and advanced behaviours of Java inner classes in the context of aliasing in a simple manner. We have shown that inner classes are sound with respect to aliasing. This is, we believe, an important result especially since, in the past, features developed in functional calculi have not remained sound when converted to a setting allowing aliasing.

$\mathcal{L}_{jm}$ should be extensible to include the other styles of inner classes, local and anonymous, since these types of inner classes do not change the host relationship, but they do have interesting behaviour with respect to other Java features, e.g. access modifiers.

We note certain similarities between the host relation and relations which arise between objects in work on ownership types. In fact [9] utilises inner classes for ownership types, and further work would be the extension of ownership types with inner classes. In that work inner classes features are not heavily exploited, though the system suggests they could be.

References

1. Walrath, K., Campione, M.: The swing tutorial (1999)
2. et al., L.M.: Object-Oriented Programming in the Beta Programming Language. First edn. (1993)
3. Igarashi, A., Pierce, B.C.: On Inner Classes. In: European Conference in Object Oriented Programming. Volume LNCS 1850. (2000)
4. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification Second Edition. Addison-Wesley, Boston, Mass. (2000)
5. Drossopoulou, S., Eisenbach, S., Khurshid, S.: Is the java type system sound? Theory and Practice of Object Systems **5** (1999) 3–24
6. Flatt, M., Krishnamurthi, S., Felleisen, M.: Classes and mixins. In: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press (1998) 171–183
7. Smith, M., Drossopoulou, S.: Inner classes visit aliasing - technical report (2003)
8. Sun Microsystems Inc.: Specification for inner classes. <http://java.sun.com/products/jdk/1.1/docs/guide/innerclasses/spec/innerclasses.doc.html> (1999)
9. Boyapati, C., Liskov, B., Shriram, L.: Ownership types for object encapsulation. In: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press (2003) 213–223

⁵ Igarashi & Pierce’s informal statement of this requirement is not strong enough, though the Java compiler is.

Java Definite Assignment in Isabelle/HOL^{*}

Norbert Schirmer

Technische Universität München
Department of Informatics
80290 Munich, Germany
Email: schirmer@in.tum.de

Abstract. In Java the compiler guarantees that each local variable is initialised when we attempt to access it at runtime. This prohibits access to uninitialised memory during execution and is a key ingredient for type safety. We have formalised the definite assignment analysis of the Java compiler in the theorem prover Isabelle/HOL and proved it correct.

1 Motivation

One potential violation of type safety during runtime is caused by uninitialised variables. An uninitialised piece of memory may contain an arbitrary sequence of bits. If we regard them as proper values and read them, we can easily produce some unpredictable behaviour. Think of an uninitialised object variable. The bit sequence is interpreted as a reference to an object in main memory and since the variable is not initialised we may read or write to an arbitrary memory location. In Java special care is taken that access to uninitialised memory does not occur. If an object is created, the fields of the object are initialised with default values. For example the default value for an object reference is `null`. For local variables the compiler and bytecode-verifier have to ensure that all variables have a *definitely assigned* value when any access to its value occurs. The terminology “Definite Assignment” stems from the Java Language Specification (JLS)[1]. The definite assignment analysis of the compiler is basically a data-flow analysis, which ensures that there is an assignment to a local variable on every possible execution path before a read of the variable content. The analysis is similar to a live variable analysis or “Definition-Use” chains [3].

The work presented in this paper is part of a comprehensive research effort aiming at formalising and verifying key aspects of the Java programming language. In particular we have a type system and an operational semantics (with a proof of type soundness) and an axiomatic semantics (with a proof of its equivalence to the operational semantics) for a large subset of Java [5,6]. All these formalisations and proofs have been carried out in the theorem prover Isabelle/HOL [4]. The definite assignment analysis was not yet part of our type system. As far as we know, the only formal model of Java source language which also treats definite assignment is the work of Stärk et al. [7]. In that work the analysis is

^{*} This research is funded by the EU project *VerifiCard*

described with data flow equations and the proofs are carried out by pencil and paper. We present the analysis as type systems. The underlying data flow analysis does not require a fixed point iteration. Therefore the presentation as type systems appears to be easier to understand as the data flow equations. Moreover our model directly implies an implementation of the analysis and provides the first machine checked formalisation. The complete sources and documentation are available online [9].

2 The definite assignment analysis

In this section we introduce a formal model of the definite assignment analysis. Due to space limitations we cannot explain all details. We only focus on the analysis of a few Java language constructs. Our aim is to illustrate how we analyse the `while` loop with all its complications, like constant conditions and `breaks`.

We define a relation written as $B \gg t \gg A$ where B is the set of initialised (or definitely assigned) local variable *before* evaluation of the Java term t and A is the set of definite assigned variables *after* its evaluation. For example for an assignment `y = x` the relation, $\{x\} \gg y = x \gg \{x, y\}$ is a valid analysis result. If we start in a state where the variable `x` was already assigned a value, then the assignment to `y` will be legal and we will end up in a state where both `x` and `y` are assigned. On the one hand the analysis will ensure that the variables we read from (like `x`) have an assigned value and on the other hand the analysis will calculate the set of variables that will be assigned after execution of the term. Let us look at a first example, taken from the JLS:

```
int k;
if (v > 0 && (k = System.in.read()) >= 0)
    System.out.println(k);
```

In this example the compiler recognises that `k` has been definitely assigned a value before it is printed out. Since the print method is only executed if the condition has evaluated to *True*, both sides of the conjunction `&&` must have been evaluated. Therefore the assignment to variable `k` was evaluated. Note that the operator `&&` only evaluates the right expression if the left one evaluated to *True*. The definite assignment analysis propagates knowledge about the outcome of a condition while analysing a branching statement. We introduce an auxiliary function *assigns-if* that calculates the set of assigned variables of a condition depending on the current branch. The semantical intuition of the term *assigns-if b e* is: “What are the assigned variables if the expression e evaluates to the boolean value b ”. In the example the function will yield the following results for the condition evaluating to *True* or *False*:

```
assigns-if True (v > 0 && (k = System.in.read()) >= 0) = {k}
assigns-if False (v > 0 && (k = System.in.read()) >= 0) = {}
```

The next example considers a **while** loop:

```
int k;
while (true) {
    k = n;
    if (k >= 5) break;
}
System.out.println(k);
```

The definite assignment rules will allow to print out **k** in this example, too. Since the condition of the **while** statement is constantly **true** the only way out of the loop is through the **break** in the body of the loop. For the analysis this means, that we have to take constant boolean conditions into account and take care of normal and of abrupt completion due to **break**. The constant value propagation in boolean conditions, to statically decide which path of evaluation will be chosen, is very basic. Only boolean literals are propagated over boolean connectives.

To properly analyse the **while** loop we need to calculate the sets of definitely assigned variables in the loop body, for both normal and **break** completion. Since **while** and **break** may also carry labels to indicate which **while** a nested **break** will complete, we have to analyse the definitely assigned variables for each of those labels. In our generalised model every **break** carries a label. In an analysis $B \gg t \gg A$ the output A is not just a set of assigned variables, but a pair of analysis results for normal and **break** completion: starting in a state where the local variables in set B are definitely assigned, if evaluation of term t completes normally, then all variables in set $nrm\ A$ are definitely assigned; if evaluation of term t completes abruptly because of a **break** with label l , then all variables in set $(brk\ A)\ l$ are definitely assigned. The analysis is designed to yield a safe approximation of the assigned variables at runtime. So whenever the analysis regards a variable as initialised, the variable will actually be initialised at the corresponding point of execution in any program run. The rules that set up the relation are defined inductively, for most terms of the language there is exactly one rule.

First we look at sequential composition written as $c_1;; c_2$ in our setting.

$$\frac{\begin{array}{cc} B \gg \langle c_1 \rangle \gg C_1 & nrm\ C_1 \gg \langle c_2 \rangle \gg C_2 \\ nrm\ A = nrm\ C_2 & brk\ A = (brk\ C_1) \Rightarrow \cap (brk\ C_2) \end{array}}{B \gg \langle c_1;; c_2 \rangle \gg A} \quad (\text{COMP})$$

The sequential composition of two statements c_1 and c_2 completes normally if both statements complete normally and completes abruptly if either of the statements completes abruptly. The second statement c_2 is only executed if the first one has completed normally. Therefore we feed the result $nrm\ C_1$ of the first part of the analysis into the second part. The result for normal completion of the second statement $nrm\ C_2$ is also the result for normal completion of the combined statement, since we know that both statements must have completed

normally. For abrupt completion either the first or the second statement completed abruptly. For each label we intersect the results of both paths. That is what the infix function $\Rightarrow \cap$ is for:

$$\begin{array}{c}
\text{def } A \Rightarrow \cap B \equiv \lambda l. A \ l \cap B \ l \\
\\
\frac{
\begin{array}{c}
B \gg \langle e \rangle \gg E \\
(B \cup \text{assigns-if } \text{True } e) \gg \langle c_1 \rangle \gg C_1 \\
(B \cup \text{assigns-if } \text{False } e) \gg \langle c_2 \rangle \gg C_2 \\
nrm A = nrm C_1 \cap nrm C_2 \quad brk A = brk C_1 \Rightarrow \cap brk C_2
\end{array}
}{B \gg \langle \text{If}(e) \ c_1 \ \text{Else } c_2 \rangle \gg A} \quad (\text{IF})
\end{array}$$

To analyse the **if** statement, we first ensure that the condition e passes the definite assignment analysis. By that we ensure that the variable accesses in e are valid, although we do not further use the output E of this analysis. We can gain a more precise approximation as input for the two conditional branches. If evaluation continues with the first branch c_1 we know that the condition e must have evaluated to *True*. In the other case, when evaluation continues with c_2 , we know that the condition e must have been *False*. This extra knowledge about the value of the condition is exploited by the function *assigns-if*. The results C_1 and C_2 of the analysis of the two possible execution paths are intersected to gain the overall result of the **if** statement. Since special care of constant values is taken by *assigns-if*, this intersection also works fine for constant branching conditions as the following example will illustrate:

```

int k;
if (true) {k = 5;} else {...}
System.out.println(k);

```

For this code fragment we get *assigns-if True true* = $\{\}$ and *assigns-if False true* = $UNIV$, where $UNIV$ is the universal set in Isabelle. We can safely regard all variables to have an assigned value in case the condition will evaluate to **false** since this will never happen. The **else** block is unreachable. This leads to $nrm C_1 = \{k\}$ and $nrm C_2 = UNIV$. Therefore, intersecting these two sets will leave us with $nrm A = \{k\}$. The same idea carries over to the **break** map as well. If we are in a path of execution that will never be reached, the set of assigned variables will be $UNIV$. When we then encounter a **break** with label l this set will be inserted in the **break** map at position l (see rule **JMP**). The lifted intersection $\Rightarrow \cap$ will work for the **break** map in the same manner as the ordinary intersection $\cap$ for the normal sets.

$$\frac{
\begin{array}{c}
B \gg \langle c \rangle \gg C \\
nrm A = nrm C \cap (brk C) \ l \quad brk A = rmlab \ l \ (brk C)
\end{array}
}{B \gg \langle l \cdot c \rangle \gg A} \quad (\text{LAB})$$

If a **break** to a label l occurs inside of a labelled statement $l \cdot c$, which carries the same label, the **break** will be absorbed and the labelled statement will complete

normally. The set of variables that are definitely assigned for normal completion then is given by the intersection of the variables for both possible paths out of c : for normal completion and for abrupt completion because of a **break** with label l . Since the labelled statement $l \cdot c$ absorbs a **break** to l it will itself never complete with such a **break**. Therefore we reset the entry for the label l in $brk\ C$ with the function $rmlab$:

def $rmlab\ l\ A \equiv \lambda\ k. \text{ if } k=l \text{ then } UNIV \text{ else } A\ k$

$$\begin{array}{c}
 nrm\ A = UNIV \\
 brk\ A = (\text{case jump of} \\
 \quad Break\ l \Rightarrow \lambda\ k. \text{ if } k=l \text{ then } B \text{ else } UNIV \\
 \quad | \text{Cont } l \Rightarrow \lambda\ k. UNIV \\
 \quad | Ret \Rightarrow \lambda\ k. UNIV) \\
 \hline
 B \gg \langle \text{Jump } jump \rangle A
 \end{array} \quad (\text{JMP})$$

The jump statements in our model combines all non exceptional abrupt completions: **break**, **continue** and **return**. Execution of a jump will never complete normally. Therefore $nrm\ A = UNIV$. Note that this implies that the definite assignment analysis will never complain about any code in sequence after the jump since this code is unreachable. The **break** map is more interesting. Since definite assignment only cares about **breaks** we can take the trivial map $\lambda\ k. UNIV$ for **continue** and **return**. In case of a **break** with label l we take the currently assigned variables B as entry for label l . For other labels we again use the trivial map.

$$\begin{array}{c}
 B \gg \langle e \rangle E \\
 (B \cup \text{assigns-if } True\ e) \gg \langle c \rangle C \\
 nrm\ A = nrm\ C \cap (B \cup \text{assigns-if } False\ e) \quad brk\ A = brk\ C \\
 \hline
 B \gg \langle l \cdot While(e)\ c \rangle A
 \end{array} \quad (\text{LOOP})$$

First of all the loop in our model of Java only handles continue jumps itself. A **break** inside the loop is handled by an enclosing label statement. A labelled loop in Java like `1: while (...) ...` is modelled by $l \cdot (l \cdot While(...) \dots)$, where the inner label is directly part of the while (to handle **continue** 1) and the outer one is an additional label statement (to handle **break** 1). Splitting up these two concepts unclutters the definite assignment analysis for the loop statement and makes the basic ideas clearer. They are the same as for the **if** statement. The loop body is analysed with the extra knowledge that the condition must have evaluated to *True*. The assigned variables for normal completion are given by the intersection of the two possible execution paths, one if we enter the body at least once and the second one if we do not enter the body at all (when the branching condition e evaluates to *False*). The analysis for constant conditions like **while** (**true**) works in the same fashion as explained for the **if** statement. Now let us look at the **break** map. Since evaluation of the branching condition will never end up in a **break** we can directly take the **break** map from the

analysis of the loop body. Let me illustrate how the rules LAB, JMP and LOOP work together to **break** out of a labelled **while**:

<code>int j;</code>		
<code>l: { while (true) {</code>	<i>nrm</i>	<i>brk</i>
<code> j = 5;</code>	$\{j\}$	$\lambda k. UNIV$
<code> break l;</code>	<i>UNIV</i>	$\lambda k. \text{if } k=l \text{ then } \{j\} \text{ else } UNIV$
<code> }</code>	<i>UNIV</i>	$\lambda k. \text{if } k=l \text{ then } \{j\} \text{ else } UNIV$
<code>}</code>	$\{j\}$	$\lambda k. UNIV$
<code>System.out.println(j);</code>		

After the assignment `j = 5` the set of definitely assigned variables that will be the input for the analysis of **break l** will be $\{j\}$. So applying the JMP rule will yield *UNIV* for normal completion and $(\lambda k. \text{if } k=l \text{ then } \{j\} \text{ else } UNIV)$ for **break** completion. We will refer to this intermediate result as *C*. Since the condition of the **while** statement is trivially **true** this will also be the result of the whole loop (since *assigns-if False true = UNIV*). Then the LAB rule has to be considered. For normal completion it will calculate $nrm C \cap (brk C) l = UNIV \cap \{j\} = \{j\}$. For completion because of a **break** it will yield *rmlab l (brk C)* $= \lambda k. UNIV$ which simply expresses that no **break** will actually leave the label statement.

Since the **break** map is important for the analysis of the **while** statement we want to take a look at how the **break** map is calculated for the **finally** statement $c_1 \text{ Finally } c_2$. Even if a **break** occurs in c_1 the block c_2 will be executed and can assign some variables. We take this into account in the analysis.

$$\begin{array}{c}
B \gg \langle c_1 \rangle \gg C_1 \quad B \gg \langle c_2 \rangle \gg C_2 \\
nrm A = nrm C_1 \cup nrm C_2 \\
brk A = ((brk C_1) \Rightarrow_{\cup} (nrm C_2)) \Rightarrow_{\cap} (brk C_2) \\
\hline
B \gg \langle c_1 \text{ Finally } c_2 \rangle \gg A
\end{array}
\quad (\text{FIN})$$

Regardless of how the first statement c_1 will complete the **finally** block c_2 will be executed. If an abrupton occurs in either statement it is (re-)raised after the completion of **finally**. If both statements terminate abruptly the second one takes precedence.

If the whole statement completes normally we can conclude that both c_1 and c_2 were executed and completed normally. Therefore we take the union of both analysis results for normal completion. In case the whole statement completes abruptly because of a **break** we regard two possibilities: Does c_2 complete normally or abruptly with a **break**. First if c_2 completes normally then the **break** has already occurred in the first statement c_1 . The assigned variables for normal completion of c_2 can be added to the **break** map of C_1 . We augment every set in the **break** map of C_1 with the set for normal completion of C_2 . This is expressed in $((brk C_1) \Rightarrow_{\cup} (nrm C_2))$ with the auxiliary operator $\Rightarrow_{\cup}$:

$$\text{def } A \Rightarrow_{\cup} B \equiv \lambda k. A \ k \cup B$$

Secondly if c_2 completes abruptly with a **break** this will show up in the overall result, because the abnormality of the second statement takes precedence. Statistically we do not know whether the first statement c_1 has completed normally or abruptly. That is why we start the analysis of c_2 with the set B assigned before the whole **finally** statement and only regard $(brk\ C_2)$ to be definitely assigned if c_2 completes with a **break**. The overall result of the **break** analysis for the **finally** statement is given by the lifted intersection of these two main paths.

3 Safe approximation

The definite assignment analysis has to be a safe approximation. This means that if the analysis will infer a variable as definitely assigned at a certain program point, then this variable will actually be assigned at that point during every execution of the program. This property is a key ingredient for type safety. Only if we know for sure that a local variable has an assigned value we can safely read this value and trust the type of the value. We have adapted the type safety proof of [5] which did not yet take definite assignment into account. To ensure type safety there, all local variables were initialised by the semantics itself. To reestablish type safety after removing this default initialisation, the key property about definite assignment we need is: the approximation of the assigned variables that the analysis yields is a subset of the variables that will actually be assigned in a program run:

theorem *da-safe-approx*:

assumes *eval*: $prg\ Env \vdash s_0 \multimap t \rightarrow (v, s_1)$

assumes *wt*: $Env \vdash t :: T$

assumes *da*: $dom\ (locals\ (store\ s_0)) \gg t \gg A$

assumes *wf*: $wf-prog\ (prg\ Env)$

shows $(normal\ s_1 \longrightarrow nrm\ A \subseteq dom\ (locals\ (store\ s_1))) \wedge$
 $(\forall l. abrupt\ s_1 = Some\ (Jump\ (Break\ l)) \wedge normal\ s_0$
 $\longrightarrow brk\ A\ l \subseteq dom\ (locals\ (store\ s_1)))$

Let me first explain the assumptions of the theorem. With *eval* we look at evaluation of a Java term t . We start in the initial state s_0 and evaluate the term t to the result value v and the final state s_1 . The semantics is an operational big step semantics. Java statements and expressions are generalised to terms in this semantics. Statements evaluate to a dummy result. With *wt* we assume that the term t is welltyped in the typing environment Env . The program component of the typing environment $prg\ Env$ describes the class and interface hierarchy and is also a parameter of the evaluation relation. The term must have passed the definite assignment analysis: $dom\ (locals\ (store\ s_0)) \gg t \gg A$. The state is decomposed into an *abrupt*- and a *store*-component. The *abrupt*-component signals all kinds of possible reasons for an abrupt completion (exceptions, **break**, **continue** and **return**). The *store*-component contains the contents of the heap and the local variables. The already assigned variables in the current state s_0 are the input variables for the definite assignment analysis. Formally we can get hold of the

already assigned variables by taking the domain (*dom*) of the local variable map *locals* of the *store* in state s_0 . Finally, assumption *wf* constrains the programs under consideration. We only look at wellformed programs. The wellformedness predicate encapsulates a lot of context conditions that we usually have in mind for a Java program. Particularly interesting for the definite assignment analysis, wellformedness ensures that all methods and static initialisers have been analysed and are welltyped. The conclusion of the theorem summarises the different aspects of definite assignment analysis that we had in mind during the design of the analysis. The main structure is a conjunction of the two constituents for normal and **break** completion:

- If evaluation completes normally (*normal* s_1) then the analysis result for normal completion *nrm* A will be a subset of the actually assigned local variables *dom* (*locals* (*store* s_1)).
- If evaluation starts in a normal state (*normal* s_0) and completes abruptly because of a **break** to label l (*abrupt* $s_1 = \text{Some } (\text{Jump } (\text{Break } l))$), the analysis result *brk* A l for the corresponding **break** set has to be a subset of the actually assigned local variables *dom* (*locals* (*store* s_1)).

The **break** part of the conclusion carries the precondition that the evaluation started in a normal state. This has to be motivated by our evaluation model used to define the operational semantics and how abrupt completion is handled. If the initial state is not normal, then abrupt completion is already signalled in the initial state and therefore the evaluation will just be skipped. The theorem is proved by induction on the evaluation relation.

4 Conclusion

In this paper we have formalised the definite assignment analysis of Java in the theorem prover Isabelle/HOL and proved that the analysis yields a safe approximation of the assigned variables at runtime. This is a key property to ensure type safe execution of a Java program. Our Java formalisation [5,6] was sufficiently mature to let us add and analyse the new concept. This again shows, that it is feasible to investigate aspects of a realistic programming language completely formally in a theorem prover. For future research it would be interesting to investigate how the source language model for definite assignment fits together with the analysis of the bytecode-verifier. The link between the source and the bytecode typesystem has already been studied [7,8]. Stärk et al. [7] have reported on some discrepancy of the definite assignment analysis of the JLS and their model of bytecode-verification. However, the bytecode-verifier presented by Klein [2] should be able to deal with it.

Acknowledgements I am grateful to Gerwin Klein, Martin Strecker and the anonymous referees for comments on this paper.

References

1. James Gosling, Bill Joy, Guy L. Steele Jr., and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison Wesley, 2000.
2. Gerwin Klein. *Verified Java Bytecode Verification*. PhD thesis, Institut für Informatik, Technische Universität München, 2003.
3. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.
4. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002. LNCS 2283.
5. David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.
6. Norbert Schirmer. Analysing the Java Package/Access Concepts in Isabelle/HOL. Technical Report NIII-R0204, Computing Science Department, University of Nijmegen, 2002.
7. Robert Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer, 2001.
8. Martin Strecker. Investigating type-certifying compilation with Isabelle. In *Proc. Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 2514 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.
9. Verificard at Munich. Available from <http://isabelle.in.tum.de/verificard>.

Flexible, source level dynamic linking and re-linking ^{*}

– Work in Progress –

Sophia Drossopoulou, Susan Eisenbach,
Imperial College London

Abstract

We give a formal semantics for dynamic linking and re-linking of code. The semantics is at source language level, and allows linking at a finer grain than current Java or C# implementations: Besides supporting the loading and verification of classes interleaved with program execution, it also allows type-safe removal and replacement of classes, fields and methods. Such extended features support unanticipated software evolution.

1 Introduction

Dynamic linking was introduced into programming languages to support implicit linking of the newest version of libraries. Partially linked code links further code on the fly, as needed, and thus, most recent updates are automatically available to end-users, without the requirement of further compilations or re-linking.

The remit has recently been extended, to support requirements from unanticipated software evolution, whereby code may need to be *updated during* program execution. To address such requirements, new features of the JVM allow the replacement of a class by a class of the same signature, as a “fix-and-continue” feature [5], while in [11] dynamic software updates support type safe dynamic reloading of code whose type may have changed, while the system is running. Also, [2] suggest dynamic linking of modules, with the support of multiple versions of the same module. Finally, [1] suggest a calculus for dynamic linking independently of the programming language and environment.

We present the semantics for a high level language similar to Java or C# with extended dynamic linking. In contrast to earlier formalizations of dynamic linking [12, 8], our model is purely at source language level. Namely, although dynamic linking is usually described at a “low level” [10], in terms of generated *.class files, its effects are visible to the source language programmer, as shown *e.g.*, through a sequence of examples in [6, 9]. This is why, in our view, an explanation at source language level is very important. In order to simplify the exposition, we do not deal with overloading or field shadowing.

In more detail, the model we suggest allows dynamic linking at a very fine grain:

DL1 Method signatures may be loaded before verification of their corresponding bodies,

DL2 Method bodies need be verified only before execution¹,

DL3 Classes may be loaded before their fields/methods,

DL4 Classes, methods and fields may be loaded “piecemeal”,

DL5 Classes, methods, and fields may be removed, if not needed,

DL6 Method bodies may be replaced.

^{*}This work partly supported by DART, EU project IST-2001-33477.

¹This is similar to C# dynamic linking.

$\frac{\chi \vdash \pi' \leq \pi \quad \mathbf{e}, \pi', \sigma, \chi \rightsquigarrow r, \pi'', \chi'}{\mathbf{e}, \pi, \sigma, \chi \rightsquigarrow r, \pi'', \chi'}$	$\frac{}{\mathbf{e}, \pi, \sigma, \chi \rightsquigarrow \text{LnkExc}, \pi, \chi}$
$\frac{\chi \ll \chi'' @ \iota \quad \iota \text{ not reachable from } \sigma \text{ in } \chi \quad \mathbf{e}, \pi, \sigma, \chi'' \rightsquigarrow r, \pi', \chi'}{\mathbf{e}, \pi, \sigma, \chi \rightsquigarrow r, \pi', \chi'}$	$\frac{}{\mathbf{y}, \pi, \sigma, \chi \rightsquigarrow \sigma(\mathbf{y}), \pi, \chi \quad \mathbf{this}, \pi, \sigma, \chi \rightsquigarrow \sigma(\mathbf{this}), \pi, \chi \quad r, \pi, \sigma, \chi \rightsquigarrow r, \pi, \chi}$
$\frac{\mathbf{e}, \pi, \sigma, \chi \rightsquigarrow \iota, \pi', \chi' \quad \mathbf{e}', \pi', \sigma, \chi' \rightsquigarrow \kappa, \pi'', \chi'' \quad \chi''(\iota)(\mathbf{f}) = \kappa', \quad \chi''' = \chi''[\iota \mapsto \chi''(\iota)[\mathbf{f} \mapsto \kappa]]}{\mathbf{e}. \mathbf{f} := \mathbf{e}', \pi, \sigma, \chi \rightsquigarrow \kappa, \pi'', \chi'''}$	$\frac{\mathbf{e}, \pi, \sigma, \chi \rightsquigarrow \iota, \pi', \chi'}{\mathbf{e}. \mathbf{f}, \pi, \sigma, \chi \rightsquigarrow \chi'(\iota)(\mathbf{f}), \pi', \chi'}$
$\frac{\mathbf{e}, \pi, \sigma, \chi \rightsquigarrow \iota, \pi_1, \chi_1 \quad \mathbf{e}_i, \pi_i, \sigma, \chi_i \rightsquigarrow \kappa_i, \pi_{i+1}, \chi_{i+1} \quad 1 \leq i \leq n \quad \chi_{n+1}(\iota) = (\mathbf{c}, \dots) \quad \mathcal{MB}(\pi_{n+1}, \mathbf{c}, \mathbf{m}) = \mathbf{e}' \quad \phi = \mathbf{this} \mapsto \iota, \mathbf{y}_1 \mapsto \kappa_1, \dots, \mathbf{y}_n \mapsto \kappa_n \quad \mathbf{e}', \pi_{n+1}, \sigma \cdot \phi, \chi_{n+1} \rightsquigarrow \kappa, \pi', \chi'}{\mathbf{e}. \mathbf{m}(\mathbf{e}_1, \dots, \mathbf{e}_n), \pi, \sigma, \chi \rightsquigarrow \kappa, \pi', \chi'}$	$\frac{\mathcal{Fs}(\pi, \mathbf{c}) = \{\mathbf{f}_1, \dots, \mathbf{f}_r\} \quad \iota \text{ is fresh in } \chi \quad \chi' = \chi[\iota \mapsto (\mathbf{c}, \mathbf{f}_1 \mapsto \mathbf{0} \dots \mathbf{f}_r \mapsto \mathbf{0})]}{\mathbf{new } \mathbf{c}, \pi, \sigma, \chi \rightsquigarrow \iota, \pi, \chi'}$

Figure 1: Execution.

DL1-DL4 represent a refinement of dynamic linking as currently implemented. **DL5-DL6** are even more “dynamic”, and come at a higher implementation cost, as they require re-checking the entire program. **DL5-DL6** have been suggested in order to support on-the-fly code replacement, and thus, unanticipated software evolution.

Furthermore, the fact that we model dynamic linking at source language level, *imposes* the requirement for **DL1** and **DL2**: Namely, when verifying a method body, $\mathbf{e}$, we need to find the signatures of any methods called within $\mathbf{e}$; for this, we load the method signatures for all called methods. But, we do *not* verify their corresponding method bodies, because if we did, we would obtain total, eager, verification *before* execution.²

Notation All mappings in our model are partial, and ϵ represents undefined. A function f' *extends* another function f at a , formally $f' \ll f @ a$, if f' is defined on one more element than f , i.e., iff $f' \downarrow_{\text{dom}(f)} = f$ and $\text{dom}(f') = \text{dom}(f) \uplus \{a\}$.

2 Syntax, runtime model, and execution

Expressions in our minimal language are method call, field assignment, field access, object creation, the receiver **this**, a parameter identifier $\mathbf{y}_i$ ($i \in \mathbb{N}$) or null indicated by **0**. Class names are indicated by $\mathbf{c}$, $\mathbf{c}'$ etc, method names are $\mathbf{m}$, $\mathbf{m}'$, etc, and field identifiers are $\mathbf{f}$, $\mathbf{f}'$.

$$\mathbf{e} \in \text{Exp} ::= \mathbf{e}. \mathbf{m}(\mathbf{e}^*) \mid \mathbf{e}. \mathbf{f} := \mathbf{e} \mid \mathbf{e}. \mathbf{f} \mid \mathbf{new } \mathbf{c} \mid \mathbf{this} \mid \mathbf{y}_i \mid \mathbf{0}$$

²This problem does not appear in Java and C# dynamic linking, because bytecode is “enriched” with the method signature for called methods; therefore, verification takes place under the assumption that the classes contain the corresponding methods, and if they do not, then an offset calculation error is thrown. In that sense, verification in our model is more eager than in Java and C#.

We represent dynamic linking through "extensions" to the underlying program π ³. Therefore, the operational semantics rewrites tuples of expressions, programs, stacks and heaps into tuples of results, programs and heaps.

$$\begin{array}{lll}
\sim & : & \text{Exp} \times \text{Prg} \times \text{Stack} \times \text{Heap} \longrightarrow \mathcal{R} \times \text{Prg} \times \text{Heap} \\
\sigma \in \text{Stack} & = & (\text{StackFrm})^* \\
\phi \in \text{StackFrm} & = & (\{\mathbf{this}\} \uplus \text{Param}) \longrightarrow \mathbb{N} \\
\chi \in \text{Heap} & = & \mathbb{N}^+ \longrightarrow \text{Obj} \\
o \in \text{Obj} & = & \text{ClsId} \times \text{FieldMap} \\
fm \in \text{FieldMap} & = & \text{FldId} \longrightarrow \mathbb{N} \\
\iota \in \mathbb{N}^+ & & \\
\kappa \in \mathbb{N} & & \\
r \in \mathcal{R} & = & \mathbb{N} \cup \{\text{nullPtrExc}, \text{StuckExc}, \text{LnkExc}\}
\end{array}$$

Stacks, σ , are sequences of stack frames, ϕ , which map **this** to an address, and the parameters $y_1, \dots, y_n$ to addresses or null. We use the shorthand $\sigma(z)$, for $\phi_n(z)$, when ϕ_n is the top of σ (*i.e.*, $\sigma = \phi_1 \dots \phi_n$). Heaps, χ , map addresses to objects. The notation $o = (c, fm)$, stands for an object of class c , with fields described by the mapping fm . For such an object, the field lookup $o(f)$ is a shorthand for $o \downarrow_2(f)$, and field update $o[f \mapsto \kappa]$ is a shorthand for $(o \downarrow_1, o \downarrow_2[f \mapsto \kappa])$. Results, r , are either addresses, or null (**0**), or exceptions. Addresses, ι , are positive numbers.

Figure 1 contains the operational semantics of the language. Rule *lnk* replaces a program by an extending program *at any time* during execution, while *exc* allows link related exceptions to be thrown *at any time* during execution. We thus have a highly non-deterministic model, similarly to [8]. Rule *gbc* allows for garbage collection – the appendix contains the definition for reachable addresses. Garbage collection is important in our study, as classes may be removed when there are no objects of that class. Garbage collection is defined in a style similar to that from [3].

The remaining rules are standard for the operational semantics of such a small object oriented language, and are similar to previous work [7]: Rule *var* describes the lookup of parameters, the receiver, or a value. Rules *fld* and *fldAss* describe field lookup and field update. Rule *call* describes method call: a new frame ϕ is pushed onto the stack, which is discarded after execution of the method body e' . Rule *new* describes object creation, where all fields of its class are initialized with **0**. In the appendix we give rules for null pointer exceptions and for stuck execution.

The different kinds of exception and the guarantees of Soundness Our calculus allows three kinds of exception: *Null pointer* exceptions, `nullPtrExc`, are thrown when a null-pointer is de-referenced to access a field or a method; they are propagated to the context as well. *Stuck* exceptions, `stuckExc`, are thrown when an object is de-referenced to access a non-existing field, or when a non-existing method is called; they are also propagated to the context. *Link* exceptions, `LnkExc`, may be thrown at any point during execution, and thus they need not be propagated to the context. They represent link-related exceptions, *i.e.*, verification errors, class-not-found errors, class-circularity-errors, load errors *etc.*⁴

Soundness, as stated in section 4, guarantees that execution will not get stuck, *i.e.*, that that execution will not attempt to access a non-existing field or method. The case of an "absent method" is slightly subtle: If the dynamic class of the receiver of a method call of m contains a method signature for that m , but contains no method body, then that represents the case where the method body could not be verified, and therefore it is a link related error. If, on the other hand, the dynamic class of the receiver contains no method signature for m , then, this corresponds to the case where the class has no such method, and therefore it is a `stuckExc`. More details are in the appendix.

³Note, that "program extensions" stands for all the dynamic linking steps from **DL1**- **DL6**, *i.e.*, even such that remove classes or methods

⁴Of course, we do not model the `FldAbsent` and `MethodAbsent` errors of Java and C#, since these are related to the "enriched" bytecode, and offset calculations.

$\frac{}{\chi \vdash \pi \leq \pi}$ <p style="text-align: center;"><i>refl</i></p>	$\frac{\chi \vdash \pi' \leq \pi'' \quad \chi \vdash \pi'' \leq \pi}{\chi \vdash \pi' \leq \pi}$ <p style="text-align: center;"><i>trans</i></p>
$\frac{\pi'_s \ll \pi_s @ c \quad \pi_s(\pi'_s(c)) \neq \epsilon}{\chi \vdash (\pi'_s, \pi_f, \pi_{ms}, \pi_{mb}) \leq (\pi_s, \pi_f, \pi_{ms}, \pi_{mb})}$ <p style="text-align: center;"><i>ldClass</i></p>	$\frac{\pi'_f(c) \ll \pi_f(c) @ f \quad \pi_s(c) \neq \epsilon \quad \chi(\iota) = (c', _) \implies \pi \not\vdash c' \leq c \quad \pi \vdash c \bowtie c' \implies \pi'_f(c)(f) = \epsilon}{\chi \vdash (\pi_s, \pi'_f, \pi_{ms}, \pi_{mb}) \leq (\pi_s, \pi_f, \pi_{ms}, \pi_{mb})}$ <p style="text-align: center;"><i>ldFld</i></p>
$\frac{\pi'_{ms} \ll \pi_{ms} @ (c, m) \quad \pi_s(\pi'_s(c)) \neq \epsilon \quad \pi \vdash c' \leq c \implies \pi \vdash \pi_{ms}(c', m) \leq \pi'_{ms}(c, m) \quad \pi \vdash c \leq c' \implies \pi \vdash \pi'_{ms}(c, m) \leq \pi_{ms}(c', m)}{\chi \vdash (\pi_s, \pi_f, \pi'_{ms}, \pi_{mb}) \leq (\pi_s, \pi_f, \pi_{ms}, \pi_{mb})}$ <p style="text-align: center;"><i>ldMthSig</i></p>	$\frac{\pi'_{mb} \ll \pi_{mb} @ (c, m) \quad \pi_{ms}(c, m) = c_1, \dots, c_{n+1}, \quad n \geq 0 \quad \pi'_{mb}(c, m) = e \quad \pi, (\mathbf{this} \mapsto c, y_1 \mapsto c_1, \dots, y_n \mapsto c_n) \vdash e : c' \quad \pi \vdash c' \leq c_{n+1}}{\chi \vdash (\pi_s, \pi_f, \pi_{ms}, \pi'_{mb}) \leq (\pi_s, \pi_f, \pi_{ms}, \pi_{mb})}$ <p style="text-align: center;"><i>verMthBdy</i></p>
$\frac{\pi_f(c) \ll \pi'_f(c) @ f \quad \pi_{ms}(c', m) = c_1, \dots, c_{n+1}, \quad \pi_{mb}(c', m) = e \implies \pi', (\mathbf{this} \mapsto c', y_1 \mapsto c_1, \dots, y_n \mapsto c_n) \vdash e : c'' \quad \pi \vdash c'' \leq c_{n+1}}{\chi \vdash (\pi_s, \pi'_f, \pi_{ms}, \pi_{mb}) \leq (\pi_s, \pi_f, \pi_{ms}, \pi_{mb})}$ <p style="text-align: center;"><i>rmFld</i></p>	$\frac{\pi_{ms} \ll \pi'_{ms} @ (c, m) \quad \pi_{mb}(c, m) = e \quad \pi_{ms}(c', m') = c_1, \dots, c_{n+1}, \quad \pi_{mb}(c', m') = e \implies \pi', (\mathbf{this} \mapsto c', y_1 \mapsto c_1, \dots, y_n \mapsto c_n) \vdash e : c'' \quad \pi \vdash c'' \leq c_{n+1}}{\chi \vdash (\pi_s, \pi_f, \pi'_{ms}, \pi_{mb}) \leq (\pi_s, \pi_f, \pi_{ms}, \pi_{mb})}$ <p style="text-align: center;"><i>rmMthSig</i></p>
$\frac{\pi_{mb} \ll \pi'_{mb} @ (c, m)}{\chi \vdash (\pi_s, \pi_f, \pi_{ms}, \pi_{mb}) \leq (\pi_s, \pi_f, \pi_{ms}, \pi'_{mb})}$ <p style="text-align: center;"><i>rmMthBdy</i></p>	
$\frac{\pi'_{mb} = \pi_{mb}[(c, m) \mapsto e] \quad \pi_{ms}(c, m) = c_1, \dots, c_{n+1}, \quad n \geq 0 \quad \pi, (\mathbf{this} \mapsto c, y_1 \mapsto c_1, \dots, y_n \mapsto c_n) \vdash e : c' \quad \pi \vdash c' \leq c_{n+1}}{\chi \vdash (\pi_s, \pi_f, \pi_{ms}, \pi'_{mb}) \leq (\pi_s, \pi_f, \pi_{ms}, \pi_{mb})}$ <p style="text-align: center;"><i>rplMthBdy</i></p>	$\frac{\pi_s \ll \pi'_s @ c \quad \chi(\iota) = (c', _) \implies \pi \not\vdash c \leq c' \quad \pi_s(c') \neq c \text{ for all } c' \quad \pi_{ms}(c, m) = \pi_f(c, f) = e \text{ for all } m, f \quad \pi_{ms}(c', m) = c_1, \dots, c_{n+1}, \quad \pi_{mb}(c, m) = e \implies \pi', (\mathbf{this} \mapsto c', y_1 \mapsto c_1, \dots, y_n \mapsto c_n) \vdash e : c'' \quad \pi \vdash c'' \leq c_{n+1}}{\chi \vdash (\pi'_s, \pi_f, \pi_{ms}, \pi_{mb}) \leq (\pi_s, \pi_f, \pi_{ms}, \pi_{mb})}$ <p style="text-align: center;"><i>rmClass</i></p>

Figure 2: Program extension

3 Programs and program extension

A program maps a class to its superclass, a class and a field identifier to its type, and class and method identifier to its signature, and method body:

$$\begin{array}{lll} \pi \in \text{Prg} = & \text{ClsId} \longrightarrow \text{ClsId} & \text{maps class to its superclass} \\ & \times \text{ClsId} \longrightarrow \text{FldId} \longrightarrow \text{ClsId} & \text{maps class and field name to type} \\ & \times \text{ClsId} \times \text{MthId} \longrightarrow (\text{ClsId}^+) & \text{maps class and meth. name to type} \\ & \times \text{ClsId} \times \text{MthId} \longrightarrow \text{Exp} & \text{maps class and meth. name to body} \end{array}$$

A program is therefore a tuple; we use the notation $(\pi_s, \pi_f, \pi_{ms}, \pi_{mb})$ to indicate the four components, and we use the naming convention that π_s , π_f , π_{ms} , and π_{mb} stand for mappings of the appropriate signatures.

We now define field and method look up functions:

$$\begin{array}{lll} \mathcal{F} & : & \text{Prg} \times \text{ClsId} \times \text{FldId} \longrightarrow \text{ClsId} \\ \mathcal{F}_s & : & \text{Prg} \times \text{ClsId} \times \text{FldId} \longrightarrow \mathcal{P}(\text{FldId}) \\ \mathcal{MS} & : & \text{Prg} \times \text{ClsId} \times \text{MthId} \longrightarrow \text{ClsId}^* \\ \mathcal{MB} & : & \text{Prg} \times \text{ClsId} \times \text{MthId} \longrightarrow \text{Exp} \end{array}$$

where, for $\pi = (\pi_s, \pi_f, \pi_{ms}, \pi_{mb})$, these functions are defined as follows:

$$\begin{aligned} \mathcal{F}(\pi, c, f) &= \begin{cases} \pi_f(c)(f) & \text{if } \pi_f(c)(f) \neq \epsilon \\ \mathcal{F}(\pi, c', f) & \text{if } \pi_f(c)(f) = \epsilon, \text{ and } c \neq \pi_s(c) = c' \\ \epsilon & \text{otherwise.} \end{cases} \\ \mathcal{F}_s(\pi, c) &= \begin{cases} \{ f \mid \mathcal{F}(\pi, c, f) \neq \epsilon \} & \text{if } \pi_s(c) \neq \epsilon \\ \epsilon & \text{otherwise} \end{cases} \\ \mathcal{MS}(\pi, c, m) &= \begin{cases} \pi_{ms}(c, m) & \text{if } \pi_{ms}(c, m) \neq \epsilon \\ \mathcal{MS}(\pi, c', m) & \text{if } \pi_{ms}(c, m) = \epsilon, \text{ and } c \neq \pi_s(c) = c' \\ \epsilon & \text{otherwise.} \end{cases} \\ \mathcal{MB}(\pi, c, m) &= \begin{cases} \pi_{mb}(c, m) & \text{if } \pi_{mb}(c, m) \neq \epsilon \\ \mathcal{MB}(\pi, c', m) & \text{if } \pi_{mb}(c, m) = \epsilon, \text{ and } c \neq \pi_s(c) = c' \\ \epsilon & \text{otherwise.} \end{cases} \end{aligned}$$

The judgments $\pi_s \vdash c' \leq c$ and $\pi \vdash c' \leq c$ mean that c' is a subclass of c , while the judgments $\pi_s \vdash c' \not\leq c$ and $\pi \vdash c' \not\leq c$ mean that the two classes are distinct with one a subclass of the other. The four judgments are defined in the appendix.

The rules in figure 2 define when a program extends another program:

refl and *trans* say that the extension relationship is reflexive and transitive.

ldCls describes loading of a class c , provided that the superclass of the loaded class has a superclass defined in the old program, i.e., for $\pi'_s(c) = c'$, we need $\pi_s(c') \neq \epsilon$. Note, that the class is loaded as an "empty shell" and its fields and methods may be loaded later, "piecemeal", through rules *ldFld* and *ldMthSig*.

ldFld describes loading the type of a field f for a class c . It requires that no objects of that class or its subclasses exist on the heap⁵, and that the field f has not been defined in any of the subclasses or superclasses of c ⁶.

⁵If we dropped this requirement, then at a later stage we could verify a method body that used the new field, although we might have objects in the heap, which did not contain that field

⁶In dynamic linking as in current JVMs and C# implementations, the situation is simpler. The requirement that there are no objects of subclass of c is implicitly guaranteed because classes are fully loaded before the first object is created, and the requirement that there is no other field f is any of the sub- or superclasses of c need not be imposed because field resolution takes the class defining the field into account.

ldMthSig describes loading the signature of a method m in a class c . It requires any signature for m in a subclass of c to be a "subsignature" of the newly loaded signature, and any signature for m in a superclass of c to be a "supersignature" of the newly loaded signature.⁷ The "subsignature judgment $\pi \vdash c'_1 \dots c'_n \leq c_1 \dots c_n$ " requires the argument types of the first to be supertypes of the corresponding argument types of the second, and the return type of the first to be a subtype of the return type of the second - or the second signature to be empty; it is defined in the appendix. Notice that it does *not* require the classes appearing in the signatures of methods or the types of fields to belong to classes that have already been loaded.

verMthBdy describes the verification of a method body. It requires the signature of the method to have already been loaded, and the body of the method to verify, *i.e.*, to be type correct, assuming that the parameters $y_1, \dots, y_n$ have types $c_1, \dots, c_n$, as prescribed by the signature, and return a result of a subtype of c_{n+1} , the return type of the signature. In figure 3 we define verification.

rmFld allows the removal of a field f from a class c , provided that no method body verification depended on the existence of that field, *i.e.*, all method bodies verify in π' , after the removal of the field. Note, that in contrast to rule *ldFld*, we do *not* require that there should be no objects of class c on the heap, since this does not affect type soundness⁸.

rmMthSig allows the removal of method signature for m for class c , provided that no method body verification depended on the existence of that method signature, *i.e.*, all method bodies verify in π' .

rmMthBdy allows the removal of method body for m for class c . Note, that this allows the removal of methods which are currently active on the stack. This is not problematic in our setting with large step operational semantics, which, in some sense, "copies" method bodies upon method call.

rplMthBdy allows replacing the method body for m in class c by expression e , provided that e satisfies the method signature.

rmClss allows the removal of class c , provided that this class is not the superclass of any other class c' , that there exist no objects of that class on the heap, that c has no fields or methods defined, and that c was not needed in the verification of any method.

Rules *ldClss*, *ldFld*, *ldMthSig* and *ldMthBody* correspond to features **DL1-DL4**, and thus, they reflect dynamic linking as in current Java and C# implementations, albeit with finer steps, thus requiring more detailed checks than current implementations. Rule *rmClss* corresponds to current JVM hotswapping.

Rules *rmFld*, *rmMthSig*, *rmMthBody*, and *rmClss* correspond to **DL5-DL6**. Because they allow the removal of entities, they make heavy requirements on checking - in effect they require re-checking the complete current program, and, in some cases, they also require scanning the complete heap. The expense of these checks is justified by the fact that runtime removal and replacement are rarely applied, but when they are applied, they are necessitated by the demand to upgrade safely a, possibly safety-critical, system on the fly.

Rules *rmFld*, *rmMthSig*, *rmMthBody*, and *rmClss* combined with rules *ldClss*, *ldFld*, *ldMthSig* and *ldMthBody* have the overall effect of replacement of entities. However, although any replacement can be represented by a sequence of such primitive removal and loading operations, there exist simple replacements, which require a long sequence of such

⁷This requirement is unnecessary in Java and C# implementations, because of dynamic linking is at byte-code level, in which method calls are annotated with the corresponding signature.

⁸This has been pointed out by the second anonymous referee!

verBasic $\frac{\pi, \gamma \vdash \mathbf{this} : \gamma(\mathbf{this}) \quad \pi, \gamma \vdash \mathbf{y} : \gamma(\mathbf{y}) \quad \pi, \gamma \vdash \mathbf{new} \ c : c \quad \pi, \gamma \vdash \mathbf{0} : c}{}$	verFldAss $\frac{\pi, \gamma \vdash e : c \quad \mathcal{F}(\pi, c, f) = c' \quad \pi, \gamma \vdash e' : c'' \quad \pi \vdash c'' \leq c'}{\pi, \gamma \vdash e.f := e' : c''}$
verFld $\frac{\pi, \gamma \vdash e : c \quad \mathcal{F}(\pi, c, f) = c'}{\pi, \gamma \vdash e.f : c'}$	verMthCll $\frac{\pi, \gamma \vdash e_i : c'_i \ 0 \leq i \leq n \quad \mathcal{MS}(\pi, c_0, m) = c_1, \dots, c_{n+1} \ n \geq 0 \quad \pi \vdash c'_i \leq c_i \ 1 \leq i \leq n}{\pi, \gamma \vdash e_0.m(e_1, \dots, e_n) : c_{n+1}}$

Figure 3: Verification.

$$\begin{array}{l}
\pi = (\pi_s, \pi_f, \pi_{ms}, \pi_{mb}) \\
\pi_s(c) = c' \neq \epsilon \implies \pi_s(c') \neq \epsilon \\
\pi_s(\mathbf{Object}) = \mathbf{Object} \\
\pi \vdash c \leq c' \text{ and } \pi \vdash c' \leq c \implies c = c' \\
\pi_f(c, f) \neq \epsilon \implies \pi_s(c) \neq \epsilon \\
\pi_{ms}(c, m) \neq \epsilon \implies \pi_s(c) \neq \epsilon \\
\pi_{mb}(c, m) \neq \epsilon \implies \pi_{ms}(c, m) \neq \epsilon \\
\pi \vdash c' \leq c, \pi_f(c', f) \neq \epsilon \implies \pi_f(c, f) = \epsilon \\
\pi \vdash c' \leq c, \pi_{ms}(c', m) \neq \epsilon \implies \pi \vdash \pi_{ms}(c', m) \leq \pi_{ms}(c, m) \\
\pi_{mb}(c, m) = e \implies \\
\pi_{ms}(c, m) = c_1, \dots, c_{n+1}, \ n \geq 0 \\
\pi, (\mathbf{this} \mapsto c, \mathbf{y}_1 \mapsto c_1, \dots, \mathbf{y}_n \mapsto c_n) \vdash e : c' \\
\pi \vdash c' \leq c_{n+1} \\
\hline
\vdash \pi
\end{array}$$

Figure 4: Well-formed programs

primitive operations *e.g.*, replacing the signature of a method which is called by other methods⁹. Therefore, although the choice of the primitive operations is attractive at a theoretical level, it is not necessarily the most natural.

4 Soundness

A program is well formed, judgment $\vdash \pi$ in figure 4, if the class hierarchy forms a tree, no field identifier is re-defined in a subclass and superclass, any method overriding a method of a superclass has signature which is a subsignature of the overridden method, and all methods bodies have a signature, and they verify to that signature. We expect to be able to prove that extension, and hence also evaluation, preserves well-formedness of the program.

A heap and stack conform to a program and environment, judgment $\pi, \gamma \vdash \chi, \sigma$ in figure 5, if the receiver and arguments point to objects which conform to their type declared in the environment. An object conforms to a type c if it is of class c' and c' is a subclass of c , and it contains appropriate values for all fields of the class c .

Soundness of the system guarantees that execution of a well typed expression in an appropriate stack and heap and in context of a well formed program will not get stuck, *i.e.*, will not

⁹This has been pointed out by the second anonymous referee!

$$\begin{array}{c}
\frac{\pi \vdash c' \leq c}{\chi(\iota) = (c', _)} \\
\frac{\pi, \chi \vdash \iota \triangleleft c}{\pi, \chi \vdash \mathbf{0} \triangleleft c}
\end{array}
\qquad
\begin{array}{c}
\frac{\chi(\iota) = (c, fm) \quad \mathcal{F}(\pi, c, f) = c' \implies \pi, \chi \vdash fm(f) \triangleleft c'}{\pi, \chi \vdash \iota} \\
\frac{\chi(\iota) \neq \epsilon \implies \pi, \chi \vdash \iota \quad \pi, \chi \vdash \sigma(\mathbf{this}) \triangleleft \gamma(\mathbf{this}) \quad \pi, \chi \vdash \sigma(y) \triangleleft \gamma(y)}{\pi, \gamma \vdash \chi, \sigma}
\end{array}$$

Figure 5: Conformance.

attempt to access non-existing fields or methods, and if it produces a value, this value is guaranteed to conform with the type of the original expression. Soundness makes no guarantees about link and pointer exceptions. We expect to be able to prove the following Theorem:

Theorem 1 (Soundness) *If*

- $\pi, \gamma \vdash \chi, \sigma, \quad \text{and} \quad \vdash \pi, \quad \text{and}$
- $\pi, \gamma \vdash e : c \quad \text{and} \quad e, \pi, \sigma, \chi \rightsquigarrow r, \pi', \chi'$

then

- $\vdash \pi', \quad \text{and} \quad \pi', \gamma \vdash \chi', \sigma,$
- $r \neq \text{StuckExc}, \quad \text{and} \quad \text{if } r = \kappa, \text{ then } \pi', \chi' \vdash \kappa \triangleleft c$

5 Conclusions and further work

We have developed a formal system which models dynamic linking and re-linking at the language level. at a very fine level. The suggested features support unanticipated software evolution.

Further work includes

- completion of the technical details and proofs,
- an investigation in how far the conditions for program extension in this paper are necessary for preservation of program well-formedness,
- a refinement of the model into a series of models reflecting more and more relevant implementation details (*e.g.*, class layout, object layout, offset calculation), and the restrictions they bring into the concept of program extension,
- an investigation in how far the restriction of Java dynamic linking (eg object creation *after loading*, *no method removal* together with our current model would lead us to the dynamic linking model as in current Java implementations,
- an investigation into ways of expressing dynamic linking operations at a coarser, or more variable granularity,
- extensions to incorporate ideas for unanticipated object reclassification, [4].

Acknowledgements We are very grateful to Giovanni Lagorio, Mariangiola Dezani and Paola Giannini, and the two anonymous FTJP referees for their feedback, suggestions, and careful checking and repairs to the formal model. Furthermore, the second anonymous TJJP referee gave insightful suggestions, and ideas on bridging the gap between theory and practical relevance.

References

- [1] Davide Ancona, Sonia Fagorzi, and Elena Zucca. A Calculus for Dynamic Linking. In *Italian Conference of Theoretical Computer Science*, 2003.
- [2] Gavin Biermann, Michael Hicks, Peter Sewell, and Gareth Stoye. Formalizing Dynamic Software Updating. In *USE Workshop ETAPS*, April 2003.
- [3] Cristiano Calcagno, Peter O’Hearn, and Richard Bornat. Program Logic and Equivalence in the Presence of Gargabe Collection. *Theoretical Computer Science*, 2002.
- [4] Ferruccio Damiani and Paola Giannini. Refined Effects for unanticipated object reclassification, Fickle₃. In *Italian Conference of Theoretical Computer Science*, 2003.
- [5] Mikhail Dimitriev. Hotspot Technology Application for Advanced Profiling. In *ECOOP USE Workshop*, June 2002.
- [6] Sophia Drossopoulou and Susan Eisenbach. Manifestations of Java Dynamic Linking. In *ECOOP Workshop on Unanticipated Software Evolution*, 2002.
- [7] Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is Java Sound? *Theory and Practice of Object Systems*, 5(1), January 1999.
- [8] Sophia Drossopoulou, Giovanni Lagorio, and Susan Eisenbach. Flexible Models for Dynamic Linking. In *12th European Symposium on Programming*, 2003.
- [9] Susan Eisenbach. Dynamic Linking phases in Java and C#, 2002. <http://www.doc.ic.ac.uk/~sue/foodexample.html>.
- [10] James Goslong, Bill Joy, Guy Steele, and Gilad Bracha. *The JavaTM Language Specification - Second Edition*. Addison-Wesley, 2000.
- [11] Michael Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic Software Updating. In *Programming Language Design and Implementation*. ACM, 2001.
- [12] Zhenyu Qian, Allen Goldberg, and Alessandro Coglio. A Formal Specification of JavaTM Class Loading. In *OOPSLA’2000*, November 2000.

Appendix

Reachable addresses An address ι *reaches* another address ι' in the context of heap χ , iff for some field f , $\chi(\iota)(f) = \iota'$, or $\chi(\iota)(f)$ reaches ι' . An address ι *is reachable* from a stack σ in χ , iff there exists a k , so that $1 \leq k \leq n$ and $\sigma = \phi_1 \dots \phi_k \dots \phi_n$, and ϕ_k (**this**) reaches ι , or $\phi_k(y)$ reaches ι for some y .

Stuck execution and null-pointer exceptions Rules *fldNull*, *fldAssNll*, and *callNull* describe null pointer exceptions. Rules *fldStck*, *fldAssStck*, and *callStck* describe stuck execution, due to a missing field or method body. Rules *propNull*, and *propStck* describe propagation of exceptions.

In particular, as we discussed in section 2, an undefined signature for the particular class, *i.e.*, $\mathcal{MS}(\dots, \dots) = \epsilon$ indicates that the class has no such method, and then execution is stuck, while a defined signature and undefined method body, *i.e.*, $\mathcal{MS}(\dots, \dots) \neg \epsilon = \mathcal{MB}(\dots, \dots)$ indicates a verification error for the method body, and then execution throws a link related exception.

$\frac{\text{fldNull}}{\frac{e, \pi, \sigma, \chi \rightsquigarrow \mathbf{0}, \pi, \chi'}{e.f, \pi, \sigma, \chi \rightsquigarrow \text{nllPtrExc}, \pi, \chi'}}$	$\frac{\text{fldAssNull}}{\frac{e, \pi, \sigma, \chi \rightsquigarrow \mathbf{0}, \pi', \chi' \quad e', \pi', \sigma, \chi' \rightsquigarrow \kappa, \pi'', \chi''}{e.f := e', \pi, \sigma, \chi \rightsquigarrow \text{nllPtrExc}, \pi'', \chi''}}$
$\frac{\text{callNull}}{\frac{e, \pi, \sigma, \chi \rightsquigarrow \mathbf{0}, \pi_1, \chi_1 \quad e_i, \pi_i, \sigma, \chi_i \rightsquigarrow \kappa_i, \pi_{i+1}, \chi_{i+1} \quad 1 \leq i \leq n}{e.m(e_1, \dots, e_n), \pi, \sigma, \chi \rightsquigarrow \text{nllPtrExc}, \pi', \chi'}}$	$\frac{\text{propNll}}{\frac{e', \pi, \sigma, \chi \rightsquigarrow \text{nllPtrExc}, \pi, \chi \quad e \text{ has subexpression } e'}{e, \pi, \sigma, \chi \rightsquigarrow \text{nllPtrExc}, \pi, \chi}}$
$\frac{\text{fldStck}}{\frac{e, \pi, \sigma, \chi \rightsquigarrow \iota, \pi, \chi' \quad \chi'(\iota)(f) = \epsilon}{e.f, \pi, \sigma, \chi \rightsquigarrow \text{stuckExc}, \pi, \chi'}}$	$\frac{\text{fldAssStck}}{\frac{e, \pi, \sigma, \chi \rightsquigarrow \iota, \pi', \chi' \quad e', \pi', \sigma, \chi' \rightsquigarrow \kappa, \pi'', \chi'' \quad \chi''(\iota)(f) = \epsilon}{e.f := e', \pi, \sigma, \chi \rightsquigarrow \text{stuckExc}, \pi'', \chi''}}$
$\frac{\text{callStck}}{\frac{e, \pi, \sigma, \chi \rightsquigarrow \mathbf{0}, \pi_1, \chi_1 \quad e_i, \pi_i, \sigma, \chi_i \rightsquigarrow \kappa_i, \pi_{i+1}, \chi_{i+1} \quad 1 \leq i \leq n \quad \chi(\iota) = (c, \dots) \quad \mathcal{MS}(\pi, c, m) = \epsilon}{e.m(e_1, \dots, e_n), \pi, \sigma, \chi \rightsquigarrow \text{stuckExc}, \pi', \chi'}}$	$\frac{\text{propStck}}{\frac{e', \pi, \sigma, \chi \rightsquigarrow \text{stuckExc}, \pi, \chi \quad e \text{ has subexpression } e'}{e, \pi, \sigma, \chi \rightsquigarrow \text{stuckExc}, \pi, \chi}}$

Subtypes The judgment $\pi_s \vdash c' \leq c$ means that c' is a (possibly indirect) subclass of c according to π_s , while $\pi_s \vdash c' \not\leq c$ means that $c' \neq c$, and either $\pi_s \vdash c' \leq c$ or $\pi_s \vdash c \leq c'$. Both judgments can be extended to full programs, giving judgments $\pi \vdash c' \leq c$ and $\pi \vdash c' \not\leq c$:

$\frac{\pi_s(c') = c}{\pi_s \vdash c' \leq c}$	$\frac{\pi_s \vdash c' \leq c'' \quad \pi_s \vdash c'' \leq c}{\pi_s \vdash c' \leq c}$	$\frac{\pi_s \vdash c' \leq c \quad \pi = (\pi_s, \pi_f, \pi_{ms}, \pi_{mb})}{\pi \vdash c' \leq c}$
$\frac{\pi_s \vdash c' \leq c \quad c \neq c'}{\pi_s \vdash c' \not\leq c}$	$\frac{\pi_s \vdash c \leq c' \quad c \neq c'}{\pi_s \vdash c' \not\leq c}$	$\frac{\pi_s \vdash c' \not\leq c \quad \pi = (\pi_s, \pi_f, \pi_{ms}, \pi_{mb})}{\pi \vdash c' \not\leq c}$

Subsignatures The judgment $\pi \vdash c'_1 \dots c'_n \leq c_1 \dots c_n$ says that the signature $c'_1 \dots c'_n$ may safely override signature $c_1 \dots c_n$ in a subclass. The first rule says that any signature may override a non existing definition, while the second rule says that subsignatures are contravariant in argument types and covariant in result types.

$\frac{}{\pi \vdash c'_1 \dots c'_n \leq c}$	$\frac{\pi \vdash c_i \leq c'_i \quad i \in 1..n-1 \quad \pi \vdash c'_n \leq c_n}{\pi \vdash c'_1 \dots c'_n \leq c_1 \dots c_n}$
--	--

Algebraic Semantics of Statements of Sequential Java

Kazem Lellahi¹, Alexandre Zamulin^{2*}

¹LIPN, URM 7030 C.N.R.S
Université Paris 13, Institut Galilée,
99, Av. J.B. Clément, 93430 Villetaneuse France
kl@lipn.univ-paris13.fr, fax +33 (0)1 4826 0712
²A.P. Ershov Institute of Informatics Systems
Siberian Division of Russian Academy of Sciences
Novosibirsk 630090, Russia
zam@iis.nsk.su, fax: +7 3832 323494

Abstract. Formal semantics of some representative expressions and statements of sequential Java in the context of the state-based algebraic model of the language proposed by the authors is given. The representative expressions are field accesses, method invocation, initialized object creation, and non initialized array creation. Both basic and composed statements are considered. Basic statements are the assignment statement, method invocation, break statement, continue statement, return statement, and throw statement. Composed statements considered are the block, if statement, while loop, and try block. Finally formal semantics of the method definition is given and a notion of program specification is introduced.

Keywords: object-oriented language, object modeling, formal semantics, Java, statements, algebraic model, ASM, state update.

1 Introduction

In this work we define formal semantics of all statements of sequential Java (this means that we do not consider synchronized statements and methods). The purpose of the whole work is to define the semantics of a programming language (Java in particular) in the style of algebraic specifications where first a signature is defined, then the set of models of the signature is defined, then it is defined in what way terms in this signature are constructed and in what way they are interpreted in a given model, and finally it is defined in what way a specification is constructed and what is the set of models of a given specification. So, we want to represent the text of a (Java) program as a specification and define the semantics of the program as a set of models of the specification.

The part of the work concerning the elaboration of notions similar to the signature, its set of models, and term (expressions) construction and interpretation for a Java program is reported in [11] an updated version of the report in a condensed form is given in [12]). The work is based on the ideas of Abstract State Machines [4] and D-oids [2].

In this paper we mainly consider the construction and interpretation of another kind of terms, Java statements, and their use in program specification. The paper is organized as follows. A short summary of our previous work is given in Section 2. Some representative Java expressions are summarized in Section 3. Representative Java statements are defined in Section 4. Formal semantics of method definition and a notion of program specification are given in Section 5. Related work is reviewed in Section 6, and concluding remarks are given in Section 7.

2 Formal model of a Java program

The Java model presented in [11] includes a type system consisting of basic types (boolean, integer, float, etc.) and reference types (interfaces, classes and arrays). The type system is used to construct a *program schema* (a counterpart of the algebra signature) consisting of class and interface declarations and

* The work of this author is supported in part by Russian Foundation for Basic Research under Grant 01-01-00787.

inheritance and implementation relations that define a subtype relation between classes and/or interfaces. The closure of the class schema is then constructed by extending each class/interface declaration by inherited fields and methods. The following notation is used in the sequel:

$\mathcal{S}$ — a program schema

$c <_{isa} c'$ — subtype relation between c and c' (class c is a subtype of c');

$x : t$ — declaration of a constant or variable with name x and type t ;

$m(r) : t[q]$ — declaration of a method with name m , string of parameter types r , result type t , and exception types q ;

$r[q]$ — declaration of a constructor with string of parameter types r and exception types q .

A program state $\mathbf{A}$ is defined as a many-sorted algebra consisting of a static part (base algebra) $\mathbf{B}$, containing sets of values and functions implementing the operations of each basic type, and dynamic part associating:

- 1) a set of references $\mathbf{A}_{ic}$ with each class/interface ic so that the set of references of a supertype includes the set of references of each its subtype,
- 2) stable constant $\mathbf{y}_{ct}^{\mathbf{A}}$ with each constant declaration $y : t$ in class c ,
- 3) updatable constant $\mathbf{x}_{ct}^{\mathbf{A}}$ with each class variable declaration $x : t$ in class c ,
- 4) partial function $\mathbf{x}_{ct}^{\mathbf{A}} : \mathbf{A}_c \rightarrow \mathbf{A}_t$ with each instance variable declaration $x : t$ in class c ,
- 5) partial functions $\mathbf{elem}_{ct}^{\mathbf{A}} : \mathbf{A}_c \times \mathbf{A}_{int} \rightarrow \mathbf{A}_t$ and $\mathbf{length}_{ct}^{\mathbf{A}} : \mathbf{A}_c \rightarrow \mathbf{A}_{int}$ with respective operations *elem* and *length* in each array type $c = array(t)$.

According to the definition of the program state, an object with reference $\mathbf{o}$ may belong to several classes in the inheritance hierarchy, and each of these classes is the type of this object. Class c is called the *most specific type* of the object $\mathbf{o}$ if $\mathbf{o}$ does not belong to any of its subclasses (it is denoted by $mst(\mathbf{o})$ in the sequel).

A program state can be transformed into another state by means of a *state update*, which is either a *function update* or *sort update*. A function update $(\mathbf{f}_{rt}, \bar{\mathbf{o}}, \mathbf{v})$ causes the function $\mathbf{f}_{rt}$ being applied to the tuple $\bar{\mathbf{o}}$ (empty tuple if $\mathbf{f}_{rt}$ is a constant) to yield the value $\mathbf{v}$ in the new state. A sort update $(c, \mathbf{o})$ causes insertion of the reference $\mathbf{o}$ into the set of the references of the class c .

An update set Γ may contain several state updates. The notation $\mathbf{A}\Gamma$ denotes the transformation of the state $\mathbf{A}$ into a new state by simultaneous firing all state updates in Γ . The closure $\bar{\Gamma}$ of an update set Γ extends Γ by all state updates needed to preserve the validity of the state algebra (e.g., a reference inserted into a sort must be inserted into all its supersorts).

A program $\mathbf{P}(\mathbf{B})$ of the schema $\mathcal{S}$ possesses a set of states $|\mathbf{P}(\mathbf{B})|$ with the same base algebra $\mathbf{B}$ and the following functions among the others:

- 1) partial function $\mathbf{m}_{cr}^{\mathbf{A}}(\mathbf{o}, \bar{\mathbf{v}})$ associated with each instance method declaration $m(r) : t[q]$ in class c and $\mathbf{A} \in |\mathbf{P}(\mathbf{B})|$ producing a pair $\langle \mathbf{A}', \mathbf{v} \rangle$ (a new algebra and value either of type t or of one of the exception types in q) for the object $\mathbf{o} \in \mathbf{A}_c$ and the tuple of arguments $\bar{\mathbf{v}}$ of the corresponding types from r ,
- 2) partial function $\mathbf{m}_{cr}^{\mathbf{A}}(\bar{\mathbf{v}})$ associated with each class (static) method declaration $m(r) : t[q]$ in class c and $\mathbf{A} \in |\mathbf{P}(\mathbf{B})|$ also producing a pair $\langle \mathbf{A}', \mathbf{v} \rangle$ for the tuple of arguments $\bar{\mathbf{v}}$ of the corresponding types from r .
- 3) partial function $\mathbf{c}_{cr}^{\mathbf{A}}(\mathbf{o}, \bar{\mathbf{v}})$ associated with each constructor declaration $r[q]$ in class c and $\mathbf{A} \in |\mathbf{P}(\mathbf{B})|$ also producing a pair $\langle \mathbf{A}', \mathbf{v} \rangle$ for the object $\mathbf{o} \in \mathbf{A}_c$ and the tuple of arguments $\bar{\mathbf{v}}$ of the corresponding types from r (a value of one of the types in q is produced in case an exception is thrown in the constructor, a value $\perp$ of type *void* in the normal case).

3 Expressions

In this section, some expressions typical of object-oriented languages are defined. For brevity, it is considered that an expression cannot throw an exception. We assume in this section that Σ_c is the internal class signature of some class c , which includes all the names than can be used in expressions and statements constructed within c (see [11] for the formal definition of Σ_c).

Syntax : Expressions involving field and method names are defined recursively by rules. They extend the usual rules of expression (term) construction, which are part of the base of recursion. To save space, we

ignore many details including access restrictions, uses of *this* and *super*, etc. (these issues are elaborated in [11, 12]). In these rules the notation $e : t$ means that expression e has type t .

For any type t , sequence of types $r = t_1, \dots, t_n$, class name s , interface/class name ic , constructor declaration $r[q]$, instance variable $x : t$ declaration, and instance method declaration $m(r) : t[q]$ accessible in Σ_c ,

$$\begin{array}{ll}
\text{(subtyping)} & \frac{ic' <_{isa} ic \quad e : ic'}{e : ic} \\
\text{(field access)} & \frac{x : t \quad e : s}{e.x : t} \\
\text{(assignments)} & \frac{x : t \quad e_l : s \quad e_r : t}{e_l.x = e_r : t} \quad \frac{x : ic \quad e_l : s}{e_l.x = null : ic} \\
\text{(method invocation)} & \frac{m : t_1 \dots t_n \rightarrow t \quad e : s \quad e_1 : t_1 \dots e_n : t_n}{e.m(e_1 \dots e_n) : t} \\
\text{(initialized object creation)} & \frac{e_1 : t_1 \dots e_n : t_n}{new s(e_1 \dots e_n) : s} \\
\text{(non initialized array creation)} & \frac{e : int}{new t[e] : array(t)}
\end{array}$$

Semantics: The semantics of expression e of type t in algebra $\mathbf{A}$ is defined by a partial function, denoted by $\llbracket e \rrbracket^{\mathbf{A}}$, producing a pair, $\langle \mathbf{A}', \mathbf{v} \rangle$ (a new algebra and value). Rules of Java state that each next argument in a list of function arguments is evaluated after the previous one and the evaluation of both the argument and the function call may change the state. In the sequel, we denote the interpretation of a sequence of argument expressions $\llbracket (e_1, \dots, e_n) \rrbracket^{\mathbf{A}}$ by a pair $\langle \mathbf{A}_{arg}, \bar{\mathbf{v}} \rangle$ where $\mathbf{A}_{arg}$ is the state produced by interpretation of e_n and $\bar{\mathbf{v}}$ the sequence $\mathbf{v}_1, \dots, \mathbf{v}_n$ of the argument values produced. We consider in the sequel that a program $P(B)$ is fixed and denote by $\mathbf{A}$ with an upper or lower index a state of this program. Given a state $\mathbf{A}$, the semantics of expressions is defined recursively by the following rules:

$$\begin{array}{ll}
\text{(subtyping)} & \frac{ic' <_{isa} ic}{\llbracket e : ic \rrbracket^{\mathbf{A}} = \llbracket e : ic' \rrbracket^{\mathbf{A}}}. \\
\text{(field access)} & \frac{\llbracket e \rrbracket^{\mathbf{A}} = \langle \mathbf{A}', \mathbf{o} \rangle}{\llbracket e.x \rrbracket^{\mathbf{A}} = \langle \mathbf{A}', \mathbf{x}_{st}^{\mathbf{A}'}(\mathbf{o}) \rangle} \quad \llbracket e.x \rrbracket^{\mathbf{A}} \text{ is undefined if so is } \mathbf{x}_{st}^{\mathbf{A}'}(\mathbf{o}) \text{ or } \llbracket e \rrbracket^{\mathbf{A}}. \\
\text{(assignments)} & \frac{\llbracket e_l \rrbracket^{\mathbf{A}} = \langle \mathbf{A}_l, \mathbf{v}_l \rangle \quad \llbracket e_r \rrbracket^{\mathbf{A}_l} = \langle \mathbf{A}_r, \mathbf{v}_r \rangle \quad \Gamma_a = \{(\mathbf{x}_s, \mathbf{v}_l, \mathbf{v}_r)\} \quad \mathbf{A}_a = \mathbf{A}_r \Gamma_a}{\llbracket e_l.x = e_r \rrbracket^{\mathbf{A}} = \langle \mathbf{A}_a, \mathbf{v}_r \rangle} \\
& \frac{\llbracket e_l \rrbracket^{\mathbf{A}} = \langle \mathbf{A}_l, \mathbf{v}_l \rangle \quad \Gamma_a = \{(\mathbf{x}_s, \mathbf{v}_l, null)\} \quad \mathbf{A}_a = \mathbf{A}_l \Gamma_a}{\llbracket e_l.x = null \rrbracket^{\mathbf{A}} = \langle \mathbf{A}_a, null \rangle} \\
& \llbracket e_l.x = e_r \rrbracket^{\mathbf{A}} \text{ and } \llbracket e_l.x = null \rrbracket^{\mathbf{A}} \text{ are not defined if so is } \mathbf{v}_r \text{ or } \mathbf{v}_l \text{ is null.} \\
\text{(instance method invocation)} & \frac{\llbracket e \rrbracket^{\mathbf{A}} = \langle \mathbf{A}_e, \mathbf{o} \rangle \quad \llbracket (e_1, \dots, e_n) \rrbracket^{\mathbf{A}_e} = \langle \mathbf{A}_{arg}, \bar{\mathbf{v}} \rangle \quad s' = mst(\mathbf{o})}{\llbracket e.m(e_1, \dots, e_n) \rrbracket^{\mathbf{A}} = m_{s'r}^{\mathbf{A}_{arg}}(\mathbf{o}, \bar{\mathbf{v}})} \\
& \llbracket e.m(e_1, \dots, e_n) \rrbracket^{\mathbf{A}} \text{ is undefined if } \llbracket e \rrbracket^{\mathbf{A}} \text{ or } \llbracket e_i \rrbracket^{\mathbf{A}_i} \text{ or } m_{s'r}^{\mathbf{A}_{arg}}(\mathbf{o}, \bar{\mathbf{v}}) \text{ is undefined.} \\
\text{(class method invocation)} & \frac{\llbracket (e_1, \dots, e_n) \rrbracket^{\mathbf{A}_e} = \langle \mathbf{A}_{arg}, \bar{\mathbf{v}} \rangle}{\llbracket s.m(e_1, \dots, e_n) \rrbracket^{\mathbf{A}} = m_{sr}^{\mathbf{A}_{arg}}(\bar{\mathbf{v}})} \\
& \llbracket e.m(e_1, \dots, e_n) \rrbracket^{\mathbf{A}} \text{ is undefined if } \llbracket e \rrbracket^{\mathbf{A}} \text{ or } \llbracket e_i \rrbracket^{\mathbf{A}_i} \text{ or } m_{sr}^{\mathbf{A}_{arg}}(\mathbf{o}, \bar{\mathbf{v}}) \text{ is undefined.} \\
\text{(initialized object creation)} & \frac{o \notin |\mathbf{A}| \quad \delta = (s, \mathbf{o}) \quad \Gamma_\delta = \{\delta\} \quad \mathbf{A}_\delta = \mathbf{A} \Gamma_\delta \quad \llbracket (e_1, \dots, e_n) \rrbracket^{\mathbf{A}_\delta} = \langle \mathbf{A}_{arg}, \bar{\mathbf{v}} \rangle \quad s_{sr}^{\mathbf{A}_{arg}}(\mathbf{o}, \bar{\mathbf{v}}) = \langle \mathbf{A}_{con}, \perp \rangle}{\llbracket new s(e_1, \dots, e_n) \rrbracket^{\mathbf{A}} = \langle \mathbf{A}_{con}, \mathbf{o} \rangle} \\
& \text{provided } \llbracket e_1 \rrbracket, \dots, \llbracket e_n \rrbracket, \text{ and } s_{sr}^{\mathbf{A}_{con}}(\mathbf{o}, \bar{\mathbf{v}}) \text{ are defined.} \\
\text{(non initialized array creation)} & \frac{\llbracket e \rrbracket^{\mathbf{A}} = \langle \mathbf{A}_e, \mathbf{n} \rangle \quad o \notin |\mathbf{A}_e| \quad \delta = (array(t), \mathbf{o}) \quad \Gamma_\delta = \{\delta\} \quad \mathbf{A}_\delta = \mathbf{A}_e \Gamma_\delta \quad \Gamma_1 = \{(\text{length}_{array(t)}, \mathbf{o}, \mathbf{n})\} \quad \mathbf{A}_1 = \mathbf{A}_\delta \Gamma_1}{\llbracket new t[e] \rrbracket^{\mathbf{A}} = \langle \mathbf{A}_1, \mathbf{o} \rangle} \\
& \llbracket new t[e] \rrbracket^{\mathbf{A}} \text{ is not defined if } \mathbf{n} < 0
\end{array}$$

In what follows we give a comment for each of the above rule.

1. A subtype expression can be used in a context where an expression of its supertype is needed.

2. A field value is produced by the corresponding field function, and a new algebra is produced in addition if the interpretation of e changes the state.
3. In the assignment expression first the left hand is evaluated and then the right hand is evaluated in the new state. The assignment (field function update) is made then producing an update set. Note that this update set is closed to ensure that the function is updated in all classes supplying or inheriting this field.
4. The interpretation of a method invocation in state $\mathbf{A}$ causes the invocation (in the state obtained by the evaluation of the method arguments) of the method associated with the most specific type of object $\mathbf{o}$. In this way, the method name is bound exactly to the method associated with the object independent of its current type, i.e., *dynamic (late) binding* is provided.
5. The interpretation of the expression $\text{new } s(t_1, \dots, t_n)$ leads to the creation of an initialized object. Note that the carrier update δ is closed to ensure the insertion of the new reference into all super-classes/superinterfaces of s .
6. A non-initialized array creation computes the length of a new array, expands the set of references of a given array type by a new reference and sets the value of the attribute *length* for this array object.

The names declared in class c may be used within the class without qualifying them by anything. The variables *this* and *super* also can be used in expressions. This is explained by the following rules. For any type t , sequence of types $r = t_1, \dots, t_n$, constructor declaration $r[q] \in \text{con}(c)$, instance variable declaration $(x : t) \in \text{ivar}(c)$, instance method declaration $(m(r) : t[q]) \in \text{imeth}(c)$ and class method declaration $(m(r) : t[q]) \in \text{cmeth}(c)$,

$$\begin{aligned}
(\text{this}) \quad & \overline{\text{this} : c} \\
(\text{super}) \quad & \frac{c <_{isa} c' \quad \nexists c'' : c <_{isa} c'' <_{isa} c'}{\text{super} : c'} \quad (\text{variable "super" gets the type of the direct superclass of } c) \\
(\text{field access}) \quad & \overline{x : t} \\
(\text{assignments}) \quad & \frac{x : t \quad e_r : t}{x = e_r : t} \quad \frac{x : ic}{x = \text{null} : ic} \\
(\text{instance method invocation}) \quad & \frac{e_1 : t_1 \dots e_n : t_n}{m(e_1 \dots e_n) : t} \\
(\text{class method invocation}) \quad & \frac{e_1 : t_1 \dots e_n : t_n}{m(e_1 \dots e_n) : t}
\end{aligned}$$

Interpretations of these expressions are very simple and do not need special comments:

$$\begin{aligned}
(\text{this}) \quad & \frac{\mathbf{o} \in \mathbf{A}_c \quad \text{this} \mapsto \mathbf{o}}{\llbracket \text{this} \rrbracket^{\mathbf{A}} = \langle \mathbf{A}, \mathbf{o} \rangle} \\
(\text{super}) \quad & \frac{\mathbf{o} \in \mathbf{A}_c \quad \text{super} \mapsto \mathbf{o}}{\llbracket \text{super} \rrbracket^{\mathbf{A}} = \langle \mathbf{A}, \mathbf{o} \rangle} \quad (\text{the subclass object with the type of its direct superclass}) \\
(\text{field access}) \quad & \overline{\llbracket x \rrbracket^{\mathbf{A}} = \llbracket \text{this}.x \rrbracket^{\mathbf{A}}} \\
(\text{assignments}) \quad & \overline{\llbracket x = e_r \rrbracket^{\mathbf{A}} = \llbracket \text{this}.x = e_r \rrbracket^{\mathbf{A}}} \quad \overline{\llbracket x = \text{null} \rrbracket^{\mathbf{A}} = \llbracket \text{this}.x = \text{null} \rrbracket^{\mathbf{A}}} \\
(\text{instance method invocation}) \quad & \overline{\llbracket m(e_1, \dots, e_n) \rrbracket^{\mathbf{A}} = \llbracket \text{this}.m(e_1, \dots, e_n) \rrbracket^{\mathbf{A}}} \\
(\text{class method invocation}) \quad & \overline{\llbracket m(e_1, \dots, e_n) \rrbracket^{\mathbf{A}} = \llbracket c.m(e_1, \dots, e_n) \rrbracket^{\mathbf{A}}}
\end{aligned}$$

4 Statements

In this section we give semantics of some representative Java statements. One can find more in [12].

Each statement is described in this paper in three parts: *definition*, *typing*, and *dynamic semantics*. The definition generally states what a well-formed statement is. The typing indicates the kind of the statement and the type of a possible result. The dynamic semantics indicates the algebra and its element that may be produced by the execution of the statement.

A statement may be provided with a label that can be used by some statements for control transfer. We consider that the label has the string type, and it is the empty string in an unlabeled statement.

Once again, we assume that Σ_c is the internal signature of some class c , which includes all the names than can be used in expressions and statements constructed within c . Note that any Σ_c -algebra is a state algebra since it contains only the elements associated with the components of the program schema. For this reason, a Σ_c -algebra is also a Σ_s -algebra for any other class s . This fact will be used in the definition of the semantics of the method definition in Section 5.

The term “ Σ -statement” used in the sequel means that the statement is constructed in the signature Σ , which is either the internal class signature or its extension in a block statement (see Section 4.2). In accordance with this, a Σ -algebra A is either a program state or some its extension (the term “state” used in the sequel refers to either of them).

Taking into account that some statements do not transfer control and some others do, we distinguish five kinds of statement: ordinary, break, continue, return and throw statements. Some of these statements do not produce any value as result, some others do. To define the typing of a complex statement, one has to analyze its behavior in different program states, and result is not always defined. Therefore, typing of a statement in state A is a partial function, denoted by $\llbracket \cdot \rrbracket^A$, which produces a pair $\langle \mathbf{t}, \mathbf{k} \rangle$, where $\mathbf{t}$ is the type of the value that may be produced by the statement and $\mathbf{k}$ is one of the symbols **O**, **B**, **C**, **R**, or **T** denoting, respectively, the *ordinary*, *break*, *continue*, *return*, or *throw* statement.

The dynamic semantics of a statement in state A is a partial function, denoted by $\llbracket \cdot \rrbracket^A$, which produces a pair, $\langle A', \mathbf{v} \rangle \in |\mathbf{P}(\mathbf{B})|_{\mathbf{t}}$, where A' is a new algebra and $\mathbf{v}$ a value of the type produced by typing. This value may be either the label of the statement to be terminated by the break statement or continued by the continue statement (empty string if no label is indicated in the corresponding break or continue statement), or the value produced by a return or throw statement (the value $\perp$ if the result type is **void**).

4.1 Basic statements

Assignment statements. If $e_l = e_r$ is an assignment expression, then $e_l = e_r$; is a Σ -statement called the *assignment statement* (i.e., the assignment expression in the context where, according to the Java grammar, a statement is needed is a statement).

Semantics. $[e_l = e_r;]^A = \langle \mathbf{void}, \mathbf{O} \rangle$ and $\llbracket e_l = e_r; \rrbracket^A = \langle A', \perp \rangle$, where A' is the algebra produced by evaluations of e_l , e_r and the assignment itself (see details in the expression section). Thus, the assignment statement is an ordinary statement not transferring control. Note that an ordinary statement always produces the result $\perp$ (a single value associated with type **void**), which means that no result is produced at all.

Method invocations. If $m(r) : \mathbf{void}[q]$, where $r = \langle t_1 \dots t_n \rangle$, is an instance method from Σ_c , and $e, e_1, \dots, e_n$ are Σ -expressions of respective types $s, t_1, \dots, t_n$, then $e.m(e_1, \dots, e_n)$ is a Σ -statement called the *instance method invocation*.

Typing: $[e.m(e_1, \dots, e_n)]^A = \langle \mathbf{t}, \mathbf{k} \rangle$.

In this tuple, $\mathbf{k}$ may be either **O** (normal termination) or **T** (exception is thrown); type $\mathbf{t}$ is **void** when $\mathbf{k}$ is **O**, and $\mathbf{t}$ is one of the types in q or a subtype of it, otherwise.

Dynamic semantics. As it was defined in Section 3, the invocation of a method on a given object is interpreted by the execution of the function associated with the method in the most specific type of the object (dynamic binding). In case of method invocation as statement, this produces a pair $\langle A', \mathbf{v} \rangle$, where A' is a new algebra and $\mathbf{v}$ a value of type t in case an exception of type t is thrown and the value $\perp$ in case of normal method termination.

The semantics of the class method invocation is similar to the semantics of the instance method invocation.

Break statements. If M is a label, then **break** and **break M** are Σ -statements called the *break statements*.

Semantics.

$$\begin{aligned} [\mathbf{break}]^A &= \langle \mathbf{string}, \mathbf{B} \rangle \text{ and } \llbracket \mathbf{break} \rrbracket^A = \langle A, \langle \rangle \rangle \\ [\mathbf{break M}]^A &= \langle \mathbf{string}, \mathbf{B} \rangle \text{ and } \llbracket \mathbf{break M} \rrbracket^A = \langle A, M \rangle. \end{aligned}$$

Thus, the typing of the statement indicates that control transfer takes place and the type of an optional label is the result type. The dynamic semantics indicates that the statement causes no state update, but it produces the label of the statement to be terminated (empty string if no label is indicated in the break

statement).

Continue statements. If M is a label, then **continue** and **continue** M are Σ -statements called the *continue statements*.

Semantics.

$$\begin{aligned} [\mathbf{continue}]^A &= \langle \mathbf{string}, \mathbf{C} \rangle \text{ and } \llbracket \mathbf{continue} \rrbracket^A = \langle A, \langle \rangle \rangle \\ [\mathbf{continue } M]^A &= \langle \mathbf{string}, \mathbf{C} \rangle \text{ and } \llbracket \mathbf{continue } M \rrbracket^A = \langle A, M \rangle. \end{aligned}$$

The semantics of this statement is similar to the previous one with the exception that another kind of statement is indicated.

Return statements. If e is a Σ -expression of type t , then **return** and **return** e are Σ -statements called the *return statements*.

Semantics.

$$\begin{aligned} [\mathbf{return}]^A &= \langle \mathbf{void}, \mathbf{R} \rangle \text{ and } \llbracket \mathbf{return} \rrbracket^A = \langle A, \perp \rangle \\ [\mathbf{return } e]^A &= \langle \mathbf{t}, \mathbf{R} \rangle \text{ and } \llbracket \mathbf{return } e \rrbracket^A = \llbracket e \rrbracket^A. \end{aligned}$$

The semantics indicates that the statement neither updates the state nor produces a value in the first case and produces a value (which becomes the result of the method containing this statement) with a possible update of the state.

Throw statements. If e is a Σ -expression of an exception type t , then **throw** e is a Σ -statement called the *throw statement*.

Semantics. $[\mathbf{throw } e]^A = \langle \mathbf{t}, \mathbf{T} \rangle$ and $\llbracket \mathbf{throw } e \rrbracket^A = \llbracket e \rrbracket^A$. Thus, the throw statement produces a result of some exception type (which will be used by the corresponding exception handler), optionally updates the state and transfers control.

4.2 Statement constructors

Complex statements are constructed recursively from basic statements by means of several statement constructors. In this section, we will define some of them. For convenience, we consider separately sequences of statements not containing local declarations and block statements containing declarations of local variables.

Sequences of statements. If $St_1, St_2, \dots, St_n$ are Σ -statements, then $\{ St_1 St_2 \dots St_n \}$ is a *sequence of Σ -statements*.

Semantics. The semantics of this statement is defined recursively.

Base case. $[\{\}]^A = \langle \mathbf{void}, \mathbf{O} \rangle$ and $\llbracket \{\} \rrbracket^A = \langle A, \perp \rangle$.

General case. Let $[St_1]^A = \langle \mathbf{t}, \mathbf{k} \rangle$ and $\llbracket St_1 \rrbracket^A = \langle A', \mathbf{v} \rangle$. If $\mathbf{k} = \mathbf{O}$ (an ordinary statement has been executed), then

$$[\{ St_1 St_2 \dots St_n \}]^A = [\{ St_2 \dots St_n \}]^{A'} \text{ and } \llbracket \{ St_1 St_2 \dots St_n \} \rrbracket^A = \llbracket \{ St_2 \dots St_n \} \rrbracket^{A'}.$$

Thus, in this case the typing of the sequence is the typing of the tail of the sequence in the new state, and the dynamic semantics depends on the execution of the tail of the sequence in the new state.

If $\mathbf{k}$ is one of $\mathbf{T}$, $\mathbf{R}$, $\mathbf{B}$, or $\mathbf{C}$ (throw, return, break or continue statement has been executed), then

$$[\{ St_1 St_2 \dots St_n \}]^A = \langle \mathbf{t}, \mathbf{k} \rangle \text{ and } \llbracket \{ St_1 St_2 \dots St_n \} \rrbracket^A = \langle A', \mathbf{v} \rangle.$$

In this case, the typing of the sequence is the typing of the first control transfer statement in the sequence, and the execution of the sequence is terminated with producing a new algebra and value.

Thus, to execute a sequence of statements starting with a state A , execute successively each next statement until either the end of the sequence is reached or a control transfer statement is encountered.

$\llbracket \{ St_1 St_2 \dots St_n \} \rrbracket^A$ is not defined if $\llbracket St_1 \rrbracket^A$ or $\llbracket \{ St_2 \dots St_n \} \rrbracket^{A'}$ is not defined.

Blocks. For brevity, we consider that only non-initialized local variables can be declared in a block, any array type used in local declarations belongs to the program schema, and declarations precede statements.

First of all, we introduce a notion of signature increment simulating declarations of local variables. Let $t_1, \dots, t_n$ be types and $x_1, \dots, x_n$ different identifiers, then the set

$$\Delta = \{x_1 : t_1, \dots, x_n : t_n\}$$

is a *signature increment*. Two signature increments

$$\Delta_1 = \{x_{11} : t_{11}, \dots, x_{1n} : t_{1n}\} \text{ and } \Delta_2 = \{x_{21} : t_{21}, \dots, x_{2m} : t_{2m}\}$$

are *union-compatible* if there is no $i = 1, \dots, n$ and $j = 1, \dots, m$ such that $x_{1i} = x_{2j}$. Thus, if a block is

declared within a block, no identifier conflict may take place (Java does not permit to redeclare in a inner block the identifiers of an outer block).

Second, we introduce a notion of block signature containing the names that can be used in the block. Let Σ_c be the internal signature of the class c and Δ_1 a signature increment, then $\Sigma_1 = (\Sigma_c, \Delta_1)$ is a *block signature*. Now, if $\Sigma_{n-1} = (\Sigma_{n-2}, \Delta_{n-1})$ is a block signature and Δ_n is a signature increment union-compatible with $\Delta_1 \cup \dots \cup \Delta_{n-1}$, then $\Sigma_n = (\Sigma_{n-1}, \Delta_n)$ is a *block signature*. Thus, a block signature generally looks as follows:

$$(((\Sigma_c, \Delta_1), \dots), \Delta_{n-1}), \Delta_n).$$

In the sequel, we denote the block signature by $\Sigma' = (\Sigma, \Delta_n)$ where $\Sigma = (((\Sigma_c, \Delta_1), \dots), \Delta_{n-1})$. We also denote $\Delta_1 \cup \dots \cup \Delta_n$ by Δ (i.e., Δ is the signature increment with respect to Σ_c).

We should recall that names declared in the class c may be used without qualification when Σ_c -expressions are constructed. Unfortunately, this is not always possible when expressions of signature Σ' , which is an extension of Σ_c , are constructed. The rules of Java state that if a local variable in a block has the same name as a field in the class, then the field name is shadowed by the local variable name. Therefore, we have to modify slightly the rules of construction of expressions using field names of the class c when they are constructed in a block and define rules for construction of expressions using local variables. For example:

- if $x : t$ is an instance variable from $\overline{ivar}(c)$ and $(x : t') \notin \Delta$ for some type t' , then x is a Σ' -expression of type t .
- if $x : t$ is a local variable from Δ , then x is a Σ' -expression of type t .

Now, if Δ_n is a signature increment, $\Sigma' = (\Sigma, \Delta_n)$, and $St_1, \dots, St_m$ a sequence of Σ' -statements, then

$$\{ \Delta_n St_1 St_2 \dots St_m \}$$

is a Σ -statement called the *block*.

Now, if $\mathbf{A}$ is a Σ -algebra, then the block algebra $\mathbf{A}'$ is constructed by extending $\mathbf{A}$ with a constant $\mathbf{x}^{\mathbf{A}'}$ of the corresponding type for each $(x : t) \in \Delta_n$, and this constant interprets the name x . Respectively, each function associated with method declaration or constructor declaration is extended so that being invoked in a block algebra, it produces another block algebra as part of the result (see [12] for details).

Semantics.

$$[\{ \Delta_n St_1 St_2 \dots St_m \}]^{\mathbf{A}} = [\{ St_1 St_2 \dots St_m \}]^{\mathbf{A}}.$$

The typing of the block is the typing of the enclosed sequence of statements. Now let

$$[\{ St_1 St_2 \dots St_m \}]^{\mathbf{A}'} = \langle \mathbf{A}', \mathbf{v} \rangle,$$

then

$$[\{ \Delta_n St_1 St_2 \dots St_m \}]^{\mathbf{A}} = \langle \mathbf{A}_1, \mathbf{v} \rangle$$

where $\mathbf{A}_1 = \mathbf{A}'|_{\Sigma}$. Thus, by taking the reduct of the algebra obtained by the execution of the block to the original signature Σ , all local variables of the block are forgotten when the block is exited.

A block with intermixed variable declarations and statements may be considered as a number of nested blocks where each of them satisfies the condition stated at the beginning of the section. If Δ_n contains an initialized local variable declaration $t x = e$, then the algebra $\mathbf{A}'$ is updated by the evaluation of the statement $x = e$.

If statements. If g is a Σ -expression of type Boolean and St and St_e are Σ -statements, then

$$\text{if } (g) St \text{ and } \text{if } (g) St \text{ else } St_e$$

are Σ -statements called the *if statements*.

Semantics. Denote **if** $(g) St$ **else** St_e by ISt and let $\mathbf{A}$ be a Σ -algebra and $\llbracket g \rrbracket^{\mathbf{A}} = \langle \mathbf{A}_g, \mathbf{g} \rangle$, then

$$\text{if } \mathbf{g} = \text{true},$$

$$\text{then } [ISt]^{\mathbf{A}} = [St]^{\mathbf{A}_g} \text{ and } \llbracket ISt \rrbracket^{\mathbf{A}} = \llbracket St \rrbracket^{\mathbf{A}_g},$$

$$\text{else } [ISt]^{\mathbf{A}} = [St_e]^{\mathbf{A}_g} \text{ and } \llbracket ISt \rrbracket^{\mathbf{A}} = \llbracket St_e \rrbracket^{\mathbf{A}_g}.$$

If $\mathbf{g} = \text{true}$, then the semantics of the statement is not defined if $\llbracket St \rrbracket^{\mathbf{A}_g}$ is not defined; if $\mathbf{g} = \text{false}$, then the semantics of the statement is not defined if $\llbracket St_e \rrbracket^{\mathbf{A}_g}$ is not defined.

The semantics of **if** $(g) St$ is self-evident.

While loops. If St is a Σ -statement and g a Σ -expression of type Boolean, then $M : \text{while } (g) St$ is a Σ -statement. The label M may be the empty string.

Semantics. Denote **while** $(g) St$ by WSt and let $g^{\mathbf{A}} = \langle \mathbf{A}_g, \mathbf{g} \rangle$. Now, if $\mathbf{g} = \text{false}$ (the execution of the statement is terminated)

then $[M : WSt]^A = \langle \text{void}, \mathbf{O} \rangle$ and $\llbracket M : WSt \rrbracket^A = \langle A_g, \perp \rangle$
else let $[St]^{A_g} = \langle \mathbf{t}_{st}, \mathbf{k}_{st} \rangle$ and $\llbracket St \rrbracket^{A_g} = \langle A_{st}, v_{st} \rangle$ in
if $\mathbf{k}_{st} = \mathbf{O}$ (St completes normally)
then $[M : WSt]^A = [M : WSt]^{A_{st}}$ and $\llbracket M : WSt \rrbracket^A = \llbracket M : WSt \rrbracket^{A_{st}}$
else (St completes abnormally)
if $\mathbf{k}_{st} = \mathbf{B}$ (break statement has been executed)
then if $(v_{st} = \mathbf{M}) \vee (v_{st} = \langle \rangle)$ (this loop should be terminated)
then $[M : WSt]^A = \langle \text{void}, \mathbf{O} \rangle$ and $\llbracket M : WSt \rrbracket^A = \langle A_{st}, \perp \rangle$ (finish normally)
else $[M : WSt]^A = \langle \text{string}, \mathbf{B} \rangle$ and $\llbracket M : WSt \rrbracket^A = \langle A_{st}, v_{st} \rangle$ (finish abnormally)
else if $\mathbf{k} = \mathbf{C}$ (continue statement has been executed)
then if $(v_{st} = \mathbf{M}) \vee (v_{st} = \langle \rangle)$ (this loop should be continued)
then $[M : WSt]^A = [M : WSt]^{A_{st}}$ and $\llbracket M : WSt \rrbracket^A = \llbracket M : WSt \rrbracket^{A_{st}}$
else $[M : WSt]^A = \langle \text{string}, \mathbf{C} \rangle$ and $\llbracket M : WSt \rrbracket^A = \langle A_{st}, v_{st} \rangle$ (finish abnormally)
else (return or throw statement has been executed)
 $[M : WSt]^A = \langle \mathbf{t}_{st}, \mathbf{k}_{st} \rangle$ and $\llbracket M : WSt \rrbracket^A = \langle A_{st}, v_{st} \rangle$ (finish abnormally).
 $\llbracket M : WSt \rrbracket^A$ is not defined if $\llbracket St \rrbracket^{A_g}$ is not defined or no definite values of typing and dynamic semantics are produced (the loop does not terminate).

The semantics of the do-while loop and for loop are defined in a similar way.

Try statements. To save space, we will consider only the general case of the try statement. Let $St, FSt, St_1, \dots, St_n$ be Σ -blocks, $p_1, \dots, p_n$ identifiers, and $t_1, \dots, t_n$ types, then

```

try  $St$ 
catch ( $t_1 \ p_1$ )  $St_1$ 
    . . .
catch ( $t_n \ p_n$ )  $St_n$ 
finally  $FSt$ 

```

is a Σ -statement called the *try statement*.

Semantics. Denote the try statement by TSt and let $[St]^A = \langle \mathbf{t}, \mathbf{k} \rangle$ and $\llbracket St \rrbracket^A = \langle A_{st}, v \rangle$. Now,

if $\mathbf{k} = \mathbf{T}$ (an exception is thrown in St)
then if t_i is the first type in the sequence $t_1, \dots, t_n$ such that $t \leq_{isa} t_i$
then let $\Sigma' = (\Sigma, \{p_i : t_i\})$, A' a Σ' -algebra extending A_{st} by the constant $p_i^{A'} = v$,
 $[St_i]^{A'} = \langle \mathbf{t}_i, \mathbf{k}_i \rangle$, $\llbracket St_i \rrbracket^{A'} = \langle A_i, v_i \rangle$, $[FSt]^{A_i} = \langle \mathbf{t}_f, \mathbf{k}_f \rangle$ and $\llbracket FSt \rrbracket^{A_i} = \langle A_f, v_f \rangle$ in:
if $\mathbf{k}_f = \mathbf{O}$ (finally block completes normally)
then $[TSt]^A = \langle \mathbf{t}_i, \mathbf{k}_i \rangle$ and $\llbracket TSt \rrbracket^A = \langle A_f, v_i \rangle$ ($v_i, \mathbf{t}_i, \mathbf{k}_i$ from St_i are used)
else $[TSt]^A = \langle \mathbf{t}_f, \mathbf{k}_f \rangle$ and $\llbracket TSt \rrbracket^A = \langle A_f, v_f \rangle$ ($v_f, \mathbf{t}_f, \mathbf{k}_f$ from FSt are used)
else (no catch clause fits t)
let $[FSt]^{A_{st}} = \langle \mathbf{t}_f, \mathbf{k}_f \rangle$ and $\llbracket FSt \rrbracket^{A_{st}} = \langle A_f, v_f \rangle$ in
if $\mathbf{k}_f = \mathbf{O}$ (FSt completes normally)
then $[TSt]^A = \langle \mathbf{t}, \mathbf{k} \rangle$ and $\llbracket TSt \rrbracket^A = \langle A_f, v \rangle$ ($v, \mathbf{t}, \mathbf{k}$ from St are used)
else $[TSt]^A = \langle \mathbf{t}_f, \mathbf{k}_f \rangle$ and $\llbracket TSt \rrbracket^A = \langle A_f, v_f \rangle$ ($v_f, \mathbf{t}_f, \mathbf{k}_f$ from FSt are used)
else (no exception thrown)
let $[FSt]^{A_{st}} = \langle \mathbf{t}_f, \mathbf{k}_f \rangle$ and $\llbracket FSt \rrbracket^{A_{st}} = \langle A_f, v_f \rangle$ in
if $\mathbf{k}_f = \mathbf{O}$ (FSt completes normally)
then $[TSt]^A = \langle \mathbf{t}, \mathbf{k} \rangle$ and $\llbracket TSt \rrbracket^A = \langle A_f, v \rangle$ ($v, \mathbf{t}, \mathbf{k}$ from St are used)
else $[TSt]^A = \langle \mathbf{t}_f, \mathbf{k}_f \rangle$ and $\llbracket TSt \rrbracket^A = \langle A_f, v_f \rangle$ ($v_f, \mathbf{t}_f, \mathbf{k}_f$ from FSt are used).

The semantics of the try block TSt is not defined if either the semantic of the internal statement St or exception handler St_i (if an exception of type t_i is thrown) or final statement FSt is not defined.

5 Method definition and program specification

If $m(r) : t[q]$, where $r = t_1, \dots, t_n$, is the declaration of a method in class c , Σ_c the internal signature of c , $\Delta = \{p_1 : t_1, \dots, p_n : t_n\}$ a signature increment, St a Σ_c -block with the signature increment Δ , and t a type, then

$t \ m(t_1 \ p_1, \dots, t_n \ p_n) \ \text{throws } q \ St$

is a *method definition* in class c . Putting in terms of algebraic specifications, the method definition can be considered as a combination of the method declaration $t \ m(t_1, \dots, t_n) \ \text{throws } q$ and the equation

$$m(p_1, \dots, p_n) = St.$$

A *program specification* $\mathcal{PS}$ for a program schema $\mathcal{S}$ is the set of definitions of the methods of $\mathcal{S}$. The program specification $\mathcal{PS}$ is *complete* if each concrete method of $\mathcal{S}$ is defined¹.

Semantics. Let Σ be the signature of the program fragment where the method m is invoked, $\mathbf{A}$ a Σ -algebra, $e, e_1, \dots, e_n$ are Σ -expressions of respective types $c, t_1, \dots, t_n$,

$$\llbracket e \rrbracket^{\mathbf{A}} = \langle \mathbf{A}_0, \mathbf{o} \rangle, \llbracket e_1 \rrbracket^{\mathbf{A}_0} = \langle \mathbf{A}_1, \mathbf{v}_1 \rangle, \dots, \llbracket e_n \rrbracket^{\mathbf{A}_{n-1}} = \langle \mathbf{A}_n, \mathbf{v}_n \rangle,$$

$\mathbf{A}_c = \mathbf{A}_n|_{\Sigma_c}$ and $\mathbf{A}'$ a (Σ_c, Δ) -algebra extending $\mathbf{A}_c$ as follows: $\mathbf{p}_1^{\mathbf{A}'} = \mathbf{v}_1, \dots, \mathbf{p}_n^{\mathbf{A}'} = \mathbf{v}_n$. Thus, using the reduct $\mathbf{A}_n|_{\Sigma_c}$, we have obtained the algebra of the signature where the block St is constructed (i.e., the algebra of the signature (Σ_c, Δ)). This algebra extends the state algebra existing at method invocation by the values of the method actual parameters². Now, let $this \mapsto \mathbf{o}$ be a valuation of the variable $this$ (it may be explicitly or implicitly used in St) and under this condition:

$$\llbracket e.m(e_1, \dots, e_n) \rrbracket^{\mathbf{A}} = \langle \mathbf{A}_m, \mathbf{v}_m \rangle, \llbracket St \rrbracket^{\mathbf{A}'} = \langle \mathbf{t}_{st}, \mathbf{k}_{st} \rangle \text{ and } \llbracket St \rrbracket^{\mathbf{A}'} = \langle \mathbf{A}_{st}, \mathbf{v}_{st} \rangle.$$

The method definition $t\ m(t_1\ p_1, \dots, t_n\ p_n)\ \text{throws}\ q\ St$ is *well-formed* iff

$\mathbf{k}_{st}$ is either $\mathbf{O}$ or $\mathbf{R}$ and $\mathbf{t}_{st}$ is `void` if t is `void`,

or

$\mathbf{k}_{st}$ is $\mathbf{R}$ and $\mathbf{t}_{st} \leq_{isa} t$ if t is not `void`,

or

$\mathbf{k}_{st}$ is $\mathbf{T}$ and there exists $t_i \in q$ such that $\mathbf{t}_{st} \leq_{isa} t_i$.

For a well-formed method definition, we define the typing of the method invocation in the following way: $\llbracket m(e_1, \dots, e_n) \rrbracket^{\mathbf{A}} = \langle \mathbf{t}_{st}, \mathbf{k} \rangle$, where $\mathbf{k} = \mathbf{O}$ if $\mathbf{k}_{st}$ is either $\mathbf{O}$ or $\mathbf{R}$ (normal termination of the method) and $\mathbf{k} = \mathbf{T}$ if $\mathbf{k}_{st}$ is $\mathbf{T}$ (exception is thrown in the method). Thus, one needs to execute the body of the method to determine its typing.

A program $\mathbf{P}(\mathbf{B})$ satisfies a method definition $t\ m(t_1\ p_1, \dots, t_n\ p_n)\ \text{throws}\ q\ St_1$ in the specification $\mathcal{PS}$ iff for any Σ where the method is invoked, any Σ -algebra $\mathbf{A}$, and any Σ -expressions $e, e_1, \dots, e_n$ of their respective types, it holds:

$$\mathbf{A}_m|_{\Sigma_c} = \mathbf{A}_{st}|_{\Sigma_c} \text{ and } \mathbf{v}_m = \mathbf{v}_{st}.$$

Note that by taking the reduct of resulting algebras to the internal class signature Σ_c we require the equivalence of program state algebras and ignore any possible block local variable. The class method and constructor definitions and their satisfaction conditions are defined in a similar way. The program $\mathbf{P}(\mathbf{B})$ is a *model* of the specification $\mathcal{PS}$ if it satisfies the definition of each method and each constructor in $\mathcal{PS}$. Thus, the set of valid programs of a given program schema consists of those ones that are the models of the specification associated with the schema. In practice, a program specification (together with its schema) is represented by a program text and the set of its models consists of all properly compiled object programs.

6 Related Work

A large amount of work concerning formal semantics of Java is done in the research project LOOP (*Logic of Object-Oriented Programming*). In the framework of this project a type-theoretic memory model of a sequential Java program is presented in [3]. A companion paper [8] reports of modeling the inheritance semantics in higher-order logic. And finally it is investigated in [10] how monads in addition to coalgebras can be used in denotational semantics of Java statements. The peculiarity of our work is definition of the program state as a many-sorted algebra and interpretation of expressions and statements in terms of state updates.

The denotational approach is also followed in [1] where actually the semantics of a Java interpreter is defined.

Operational semantics of almost complete Java is defined in several works [7, 5, 13]. Once again, an abstract interpreter of a Java program is defined in each of these works. Consider as an example the

¹ In fact, a program specification may also contain static and dynamic initializers and the initializers of variables and constants (i.e., anything that can be present in a real program text), but we do not discuss them in this paper for brevity.

² Recall that any Σ_c -algebra is also a Σ_s -algebra for any other class s . Therefore, if Σ is the signature of some block where method m is invoked, then the reduct permits us to "forget" the local variables of the block and correctly interpret the symbols in the method body St .

definition of Java by Abstract State Machines in [13]. The state of the interpreter is defined there by a large number of so called dynamic functions (such a function can be different in different states). Each time a particular program construction is parsed, the corresponding dynamic functions are updated, and control passes to the "next task" defined by the dynamic function "task" (a counterpart of object program instruction counter). In contrast to this, we define an abstract program model and use its components as denotations for Java constructions. Our approach helps to avoid a large quantity of auxiliary functions representing the interpreter's state and thus make the semantics more concise and clear.

Several small sublanguages of Java are designed to experiment with semantics suitable for a theorem prover like Isabelle/HOL, while our primary goal is to experiment in providing an abstract model naturally simulating the Java program.

7 Conclusion and future work

We have introduced a formal model of a sequential Java program with states as algebras and have shown how some Java statements can be constructed and interpreted in terms of this model. The novelty of the approach consists in defining a specialized high-level operational machine whose components naturally represent program components. This provides us with a clear way of defining the semantics of an imperative language: basing on the type system of the language, define a program schema and an operational machine for this schema, then give the interpretations of expressions and statements in terms of this machine and use them for method specifications. Java has been chosen in this work to illustrate that even such complex features of the language as interfaces in addition to classes, implementation in addition to inheritance, object identity, late binding, etc. can be easily modeled with the use of simple algebraic notions including sets and functions.

Because of lack of space, several interesting aspects of the model have not been presented in this paper, local classes are among them. Particularly interesting is the task of using the presented approach for defining a state-based model of a parallel Java-program. We are proceeding with research in these directions.

Finally, we thank the anonymous reviewers for the insightful comments on the submitted text of the paper.

References

1. J. Alves-Foss and F. S. Lam. Dynamic Denotational Semantics of Java. *Formal Syntax and Semantics of Java*, LNCS, vol. 1523, 1999, pp. 201-240.
2. E. Asteziano and E. Zucca, *D-oids: a Model for Dynamic Data Types*, Mathematical Structures in Computer Science, 5(2), June 1995, 257-282.
3. J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A Type-Theoretic Memory Model for Verification of Sequential Java Programs *14th International Workshop on Algebraic Development Techniques (WADT'99)*, LNCS, vol. 1827, 2000.
4. Egon Börger and Robert Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*, Springer-Verlag, 2003.
5. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An Event-Based Structural Operational Semantics of Multi-Threaded Java. *Formal Syntax and Semantics of Java*, LNCS, vol. 1523, 1999, pp. 157-200.
6. J. Cosling, B. Joy, G. Steele, and G. Bracha. *The JavaTM Language Specification (Second Edition)*. Addison-Wesley, 2000.
7. S. Drossopoulou and S. Eisenbach. Describing the Semantics of Java and Providing Type Soundness. *Formal Syntax and Semantics of Java*, LNCS, vol. 1523, 1999, pp. 41-82.
8. M. Huisman and B. Jacobs. Inheritance in Higher Order Logic: Modeling and Reasoning. *Theorem Proving in Higher Order Logics (TPHOL)*, LNCS 1869, 2000, p.301-319.
9. B. Jacobs. Objects and Classes, Coalgebraically. *Object Orientation with Parallelism and Persistence*, Kluwer Academic Publishers, 1996, pp. 83-103.
10. B. Jacobs and E. Poll. Coalgebras and Monads in the Semantics of Java. *Theoretical Computer Science*, 2002, 291/3 p.329-349.
11. K. Lellahi, A. Zamulin. *Implicit State Approach for Formalization of Sequential Java-like Programs*. Research report of LIPN 2002-10, Univ. Paris 13, 2002 (http://www.iis.nsk.su/persons/zamulin/LZ_report.zip).
12. K. Lellahi, A. Zamulin. *State-based Algebraic Semantics of Statements of Sequential Java*. Research report of LIPN 2003-03, Univ. Paris 13, 2003
13. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.

JML Support for Primitive Arbitrary Precision Numeric Types: Definition and Semantics

Patrice Chalin

Computer Science Department, Concordia University
www.cs.concordia.ca/~faculty/chalin

1 Introduction

The Java Modeling Language, JML, is a notation for specifying and describing the detailed design and implementation of Java modules. It is a model-based specification language offering, in particular, method specification by pre- and post-condition, and class invariants to document required module behavior. It has recently been noted that the JML semantics of expressions over numeric types do not correspond to user expectations. As a result, an unusually high number of published JML specifications are invalid or inconsistent, including cases from the security critical area of smart card applications [Chalin03a]. In this extended abstract we briefly describe JML’s ancestry and language design principles (Section 2). This will help to explain the origin of the “semantic gap” between user expectations and the current meaning given to JML numeric expressions. With the objective of better matching user expectations, we introduce JMLa, a variant of JML supporting *primitive arbitrary precision* numeric types as well as implicit promotion to these types (Section 3). This is done in a manner that is consistent with JML’s language design goals and objectives [Chalin03a]. A preliminary formal semantics of JMLa expressions is given (Section 4) as well as an example of its application. Related and future work are described (Sections 5 and 6, respectively).

2 JML

2.1 Ancestry and language design principles

JML is a Behavioral Interface Specification Language (BISL). By definition, a BISL is tightly coupled to a particular programming language since its purpose is to allow developers to specify modules written in that programming language. A behavioral interface specification is a description of a module consisting of two main parts [Wing87]:

- an *interface*, that captures language specific elements that are exported by the module, such as field and method signatures;
- a *behavior*—as well as other properties and constraints—of the elements described in the interface.

Prior to JML, the main BISLs were members of the Larch family of languages of which two notable members are Larch/C++ [Leavens99] and LCL, the Larch/C interface specification language [GH93]. A key characteristic of Larch is its two-tiered approach. The *shared* tier contains specifications written in the *Larch Shared Language* (LSL). These shared tier specifications, called traits, define multisorted first-order theories. The *interface* tier contains specifications written in a Larch interface language. Each interface language is specialized for use with a particular programming language, but all interface languages make use of LSL to express module behavior [GH93]. In a departure from the Larch tradition, Leavens *et. al.* have defined JML as a single-tier BISL [LBR02]. Experience with Larch/C++ lead to the opinion that having to learn two—somewhat disparate—languages (C++ and LSL) in order to be able to read and write specifications, was too big a hurdle to overcome for most developers. The design intent has been to make JML a superset of (a significant subset of) Java. A key language design principle of JML has been to preserve the semantics of Java to the extent possible: that is, if a phrase is valid in Java and JML, then it should have the same meaning in both languages. Adherence to this principle greatly reduces the burden required to learn, understand and use JML.

```

/*@ public normal_behavior
@   requires y >= 0;
@   ensures Math.abs(\result) <= y
@   && \result * \result <= y
@   && y < (Math.abs(\result) + 1)
@       * (Math.abs(\result) + 1);
@*/
public static int isqrt(int y)

```

Figure 1. JML specification of `isqrt(int)`

```

/*@ spec_public */
private short intPart, decPart;

/*@ normal_behavior
@   modifiable intPart, decPart;
@   ensures      intPart == -\old(intPart) &&
@               decPart == -\old(decPart) ...;
@*/
public Decimal oppose()

```

Figure 2. `Decimal` class specification excerpt

As is often the case for language design principles, its benefits come at a cost. Java was designed as a programming language, not a specification language. Although JML builds upon Java by adding language constructs for the purpose of expressing specifications, it remains that core Java phrase sets, like expressions, are (for the most part) shared by both languages. This renders expression semantics more complex than, for example, in Larch. Furthermore, as we shall see in the following section, developers are in a different mindset when reading or writing specifications, particularly when it comes to reasoning about integer arithmetic.

2.2 A semantic gap, motivating examples

Consider the specification in Figure 1 of an integer square root method, `isqrt`; it was excerpted from the June 2002 edition of the main JML reference document [LBR02]. The specification requires that a caller invoke the method with a nonnegative argument y , and in return, the method ensures that it will yield a resulting value, r , such that: $|r| \leq y \wedge r^2 \leq y < (|r| + 1)^2$. The current definition of JML states that the expressions in the `requires` and `ensures` clauses of Figure 1 are to be interpreted using the semantics of Java. As a consequence (and a simple Java prototype will justify this claim), a valid implementation of `isqrt` would be permitted to return `Integer.MIN_VALUE` when y is 0. This unexpected situation arises because Java integral types have a fixed precision and because operators over these types obey rules similar to modular arithmetic—thus, for example `Integer.MIN_VALUE = Integer.MAX_VALUE + 1`. As another example consider the specification, given in Figure 2, that has been excerpt from a paper on the formal verification of an electronic purse applet [BvdBJ02]. In this case, the specification is inconsistent (it is unsatisfiable when, say, `intPart` is `Short.MIN_VALUE`).

What has gone wrong? These JML specifications (and others) demonstrate that specifiers most often ignore the finiteness of numeric types. Stated positively, specifiers generally *think* in terms of arbitrary precision arithmetic when they read and write specifications. A survey is given in [Chalin03a] of invalid and inconsistent JML specifications caused by this problem (including the two just described). Hence, there is a semantic gap between user expectations and the current language design and semantics of JML numeric types. Our attempts at mending this gap without changing the language definition of JML have been unfruitful [Chalin03b]. In particular, use of the `JMLInfiniteInteger` model class yields specifications that are overly verbose and hence difficult to read and understand. With the objective of closing the semantic gap, we propose in the next section a variant of JML named JMLa.

3 JMLa

Most JML specifiers think in terms of arbitrary precision arithmetic, yet the semantics of expressions in JML is such that fixed precision arithmetic is the *default* interpretation. We have defined a variant of JML called JMLa that offers support for the arbitrary precision numeric types `\bigint` and `\real` [Chalin03a]. The semantics of JMLa ensure that numeric operations that can cause arithmetic overflow are performed over *arbitrary precision types* by default. We will call the operators that can result in overflow *unsafe* operators; they are: unary `-`, binary `+`, `-`, `*` and `/`. Generally, in JMLa, unsafe operators will promote their integral operands to `\bigint`. As an exception to this rule, we preserve the semantics of Java for constant expressions whose evaluation would not result in an overflow. Examples are given in Figure 3. Notice how `-5` has type `int` whereas `-Integer.MIN_VALUE` has type `\bigint` because evaluation of the latter constant expression in Java would result in an overflow.

JMLa Expression	Equivalent JMLa Expression (all conversions made explicit)	Result Type	Same semantics as in Java?
<code>+i</code>	<code>+i</code>	<code>int</code>	YES, since unary <code>+</code> is safe.
<code>-i</code>	<code>-(\bigint)i</code>	<code>\bigint</code>	No, unary <code>-</code> is unsafe.
<code>-5</code>	<code>-5</code>	<code>int</code>	YES, since <code>-5</code> is an <code>int</code> value.
<code>-Integer.MIN_VALUE</code>	<code>-(\bigint)Integer.MIN_VALUE</code>	<code>\bigint</code>	No, since the constant expression value is not in the range of <code>int</code> .
<code>i + j</code>	<code>(\bigint)i + (\bigint)j</code>	<code>\bigint</code>	No, since binary <code>+</code> is unsafe.
<code>Integer.MIN_VALUE - 1</code>	<code>(\bigint)Integer.MIN_VALUE - (\bigint)1</code>	<code>\bigint</code>	No, since the expression value is not in the range of <code>int</code> .
<code>2 + k</code>	<code>(\bigint)2 + k</code>	<code>\bigint</code>	No, since <code>\bigint</code> is not a Java type.
<code>2 + (int)k</code>	<code>(int)2 + (int)k</code>	<code>int</code>	Almost.
<code>i * f</code>	<code>(\real)i * (\real)f</code>	<code>\real</code>	No. Note that a floating-point type causes promotion to <code>\real</code> .
<code>3 * 5 - Short.MAX_VALUE</code>	<code>3 * 5 - (int)Short.MAX_VALUE</code>	<code>int</code>	YES (const. expr. value is an <code>int</code>).
<code>d / 0.5</code>	<code>(\real)d / (\real)0.5</code>	<code>\real</code>	No.
<code>(\bigint)d / 2</code>	<code>(\bigint)d / (\bigint)2</code>	<code>\bigint</code>	No.

Figure 3. Sample JMLa expressions (assume `int i,j`; `\bigint k`; float `f`; double `d`)

Narrowing casts of primitive numeric types are also given a special semantics in JMLa. For example, if `i` is an `int`, then `(byte)i` is interpreted as `narrowToByte(i)` where this method is defined as:

```
static byte narrowToByte(int i) {
    if((byte)i != i) {
        throw new ArithmeticException("Value out of range for type byte: " + i);
    }
    return (byte)i; // no loss of precision.
}
```

On occasion, we need to write specifications in which some of the arithmetic operators are to be interpreted with their Java semantics. This occurs, for example, when documentation existing API classes (e.g. `java.lang.*`). For this purpose, we define the semantics of the JMLa expression `\nowarn(E)` to be the same as the Java semantics of *E*. Within `\nowarn(E)` one can make use of `\warn()` to re-enable JMLa semantics. As a variant, `\nowarn_op(E)` allows us to specify that only the outer-most operator in *E* should be interpreted with Java semantics. Similarly, `\warn_op(E)` only re-enables JMLa semantics for the outer-most operator in *E*. Hence, the JMLa expression `\nowarn_op((byte)E)` has the effect of interpreting *E* with JMLa semantics and then, like in Java, silently casting the result to `byte`. As another example we note that `\nowarn(Integer.MAX_VALUE + 1) == Integer.MIN_VALUE` would evaluate to `true`.

For the sake of conciseness in this extended abstract, we simply enumerate some of the key advantages of JMLa over JML—justifications of these claims as well as further details are provided in [Chalin03a]:

- JMLa semantics more closely match user expectations. We demonstrate in [Chalin03a] how all of the invalid or inconsistent JML specifications given in that report recover their validity and consistency when interpreted under JMLa with little or no changes to the specifications.
- JMLa can be used to write simpler, and clearer specifications (as compared, e.g., to use of `JMLInfiniteInteger`).
- The meaning of JMLa specifications can be independent of the particular choice of numeric types of fields and variables (as it should be since, e.g., method specifications are meant to express *essential* method behavior which often is independent of field and variable types).¹
- ESC/Java [Flanagan+02] will be able to detect more errors under JMLa semantics than it currently can for JML.
- Verification proofs will be greatly simplified as we recover the familiar laws of arithmetic when operating over `\bigint` and `\real` (e.g. associativity, commutativity and closure of operators).

¹ This is not the case in Java and JML; e.g. “`\result == E`” may be unsatisfiable if `\result` is declared to be of type `short`.

$$\begin{aligned}
e \in \text{EXPR} &::= c_\tau \mid \iota \mid op(e_1, \dots, e_k) \mid (\tau) e \mid \backslash\text{old}(e) \mid e.\iota \mid e.\iota(e_1, \dots, e_k) \mid (q \text{ tp } \iota; e) \\
&\quad \mid \backslash\text{nowarn_op}(e) \mid \backslash\text{warn_op}(e) \mid \backslash\text{nowarn}(e) \mid \backslash\text{warn}(e) \mid \dots \\
\tau \in \text{TYPERNM} &::= \iota \mid \dots \\
op \in \text{OPNM} & \\
q \in \text{QUANTNM} &::= \backslash\text{forall} \mid \backslash\text{exists}
\end{aligned}$$

Figure 4. Abstract syntax of JMLa expressions

These points are particularly important as we witness the increased use of JML, especially in security critical areas like smart cards. Of course, these benefits come at the cost of a slightly more complex semantics and an increased departure from Java semantics. We believe though, that the benefits of JMLa outweigh its disadvantages.

4 JMLa Semantics

This section offers a glimpse at work in progress on the formalization of JMLa semantics by means of an embedding into the language of PVS [PVS]. PVS is a specification language integrated with a theorem prover. The specification language of PVS is based on classical, typed higher-order logic. For simplicity we can assume that each JMLa specification is translated into a PVS theory. JMLa expressions are translated into PVS expressions. In this extended abstract we focus on the semantics of expressions over primitive numeric types while ignoring the very important issue of abnormal termination in expressions.

4.1 Abstract syntax and semantic objects

The semantics of JMLa expressions is defined by means of an “inference system” in a style referred to as natural semantics [Winskel93]. The inference rules allow us to establish the validity of *elaboration predicates* of the form

$$\rho \vdash a \xrightarrow{A} x$$

where A is generally the name of an abstract syntax phrase class. Such a predicate asserts that the syntactic object a corresponds to the semantic object x under the context ρ ; we will also say “ a elaborates to x under ρ .” For the cases covered here, the context will be an environment containing the declarations under which elaboration is to be performed. Furthermore, a will be a JMLa expression and x a PVS expression qualified with its type.

The abstract syntax for expressions relevant to our presentation is given in Figure 4. The cases defined are:

- An integral literal constant of type $\tau \in \{\text{int}, \text{long}\}$.
- An identifier representing a logical variable (including `\result` and method parameters)².
- An operator applied to one or more arguments. Operators include those of Java (e.g. `+`, `-`, `*`) and JML (e.g. `==>`, `<==>`).
- A type cast expression.
- A pre-state expression which is of the form `\old(e)`.
- A field access expression.
- A method invocation expression. (Recall that methods in JML expressions must be “pure” [LBR02].)
- A quantified expression.
- Warn/nowarn expressions.

JMLa expressions are translated into the “semantic objects” of PVS expressions, whose *annotated* abstract syntax is

$$\epsilon \in \text{PVSEXPR} ::= c : \tau \mid op(\epsilon_1, \dots, \epsilon_k) : \tau$$

² Since our focus in this extended abstract is on the particularities of JMLa semantics of expressions over numeric types, we shall make the simplifying assumption that all class and instance members are expressed in the form $e.\iota$ so as to be distinguishable from the occurrence of a logical variable.

```

int: THEORY
BEGIN
...
twoPn: posint = 4294967296
max : nat    = 2147483647
min  : negint = -2147483648

int: TYPE+ = {i: integer | min <= i AND i <= max} CONTAINING 0;

% narrowing primitive conversion to int
narrow(i: integer): int =
  LET b:nat = mod(i, twoPn) IN
  IF b <= max THEN b ELSE b - twoPn ENDIF

%-----
neg(i:int): int = narrow(-i)
add(i,j:int): int = narrow(i + j)
sub(i,j:int): int = narrow(i - j)
mul(i,j:int): int = narrow(i * j)
div(i:int, j:{j:int | j /= 0}): int = narrow(div.div(i,j))
...
%-----
bit_neg(i:int): int = -i - 1
...
END int

```

Figure 5. PVS theory for `int`

Each PVS expression is annotated with its type. This allows us to ensure that, in particular, overloaded operators can be disambiguated. Elaboration of expressions is done in the context of an environment, $\rho \in \text{ENV}$ that can be thought of as a mapping from identifiers³ into their attributes. The updated environment denoted by $\rho \oplus \{ \iota \mapsto \alpha \}$ is the same as ρ except that it maps ι to α . Note that in the initial JMLa environment ρ_0 , `\safeMath` is `true`.

4.2 Primitive numeric types in PVS

Before presenting the elaboration rules, let us explain how JMLa primitive numeric types are modeled in PVS. The JMLa arbitrary precision types `\bigint` and `\real` are modeled by the standard PVS types `integer` and `real`. For convenience, we have also defined a synonym for `integer` named `bigint`. We have created simple theories, all of the same form, for each of the bounded precision integral types. As an example, an excerpt of the theory for `int` is given in Figure 5. Notice how the `int` type is simply defined as the subtype of `integer` that contains values in the range `min` to `max` inclusive. A key function in this theory is `narrow` which effectively defines narrowing primitive conversion to `int`. All arithmetic operators are defined using their `integer` counterparts followed by an application of `narrow`. Thus, addition of `int`'s is defined as the addition of their values interpreted as `integer`'s followed by a narrowing of the result to `int`: i.e. `add(i,j) = narrow((i:int + j:int):integer):int`.

4.3 Elaboration rules

A simplified version of the elaboration rules for JMLa expressions is given in Figure 8 on page 9. For each syntactic case of `EXPR` we have combined the type and expression elaborations into a single rule. Aside from the “\” in JMLa type names `\bigint` and `\real`, JMLa and PVS type names coincide, hence we will make no distinction between them, using the same type name τ in both the abstract syntax and PVS expressions. This given simplified semantics does not cover rules for constant expressions, and (as was mentioned earlier) it ignores issues of abnormal termination.

The rules for literals [Literal] and logical variables [Logical Var] illustrate that they are translated almost literally. Thus `3` and `5L` elaborate to `3:int` and `5:long` respectively. Similarly, provided that a logical variable has been declared in scope, it elaborates to the same name qualified with its declared type⁴.

³ Including special identifiers like `\state`, denoting the default PVS state variable context, and `\safeMath`, `\result`, etc.

⁴ Of course `\result` would have to be mapped to a special PVS name.

Operator(s) op	Argument Type(s) $ArgType_{op}$	Argument Conversion $conv_{op}$	Result Type $resultType_{op}$	PVS Function(s) ϕ_{op}
$+$ (unary)	Numeric ⁵	unp to τ	τ	$id: \tau \rightarrow \tau$
$-$ (unary)	Numeric	unp to τ	τ	$\tau.neg$
$\sim$	Integral	unp to τ	τ	$\tau.bit_neg$
$!$	boolean	none	boolean	NOT
$*, /, \%$	Numeric, Numeric	bnp to τ	τ	$\tau.mul, \tau.div, \tau.rem$
$+, -$	Numeric, Numeric	bnp to τ	τ	$\tau.add, \tau.sub$
$<<, >>, >>>$	Integral, Integral	arg1: unp to τ arg2: unp	τ	$\tau.lshift, \tau.rshift,$ $\tau.rshift_u$
$<, <=, >, >=$	Numeric, Numeric	bnp	boolean	$<, <=, >, >=$
$==, !=$	Numeric, Numeric	bnp	boolean	$==, /=$
	boolean, boolean	none	boolean	$==, /=$
$\&, \wedge, $	boolean, boolean	none	boolean	AND, XOR, OR
	Integral, Integral	bnp to τ	τ	$\tau.bit_and, \tau.bit_xor,$ $\tau.bit_or$
$\&\&, $	boolean, boolean	none	boolean	AND, OR
$==>$	boolean, boolean	none	boolean	$==>$
$<==>, <!=>$	boolean, boolean	none	boolean	$==, /=$

Table 1. Semantics of selected JMLa operators

There are two elaboration rules for expressions involving operators. $[Op_{safe}]$ applies to all unsafe operators (these are listed in Section 3) while in “safe math” mode. JMLa specifications are interpreted in safe math mode by default. This default can be changed by the use of operators `\nowarn`, `\warn`, etc., as can be seen in the Warn/Nowarn rules. The $[Op]$ rule applies to operators that are not unsafe, or to any operator while in unsafe math mode. To apply either of these rules one must make use of the information provided in Table 1. Let op be an operator that appears in column 1 of the table, then the remaining columns define: the required argument type(s) ($ArgType_{op}$), the kind of argument conversion to be applied ($conv_{op}$), the resulting type of the expression ($resultType_{op}$) and finally, the PVS function corresponding to the operator op (ϕ_{op}). The argument conversions that can be applied correspond to unary numeric promotion or binary numeric promotion. These promotion functions come in two variants (see Figure 6): those corresponding to implicit promotion to arbitrary precision types ($JML.*$) and those that imitate the standard Java promotion rules ($Java.*$).

Type casts are processed in two cases depending on the nature of the cast: i.e. whether it corresponds to a widening [Widen] or narrowing primitive conversion [Narrow]. Type widening requires no special operator in PVS. On the other hand, narrowing to type τ requires the application of the `narrow` function defined in the theory of τ .

The remaining rules are further discussed in [Chalin03b].

⁵ Recall that for JMLa, primitive numeric types include `\real` and `\bigint`.

<pre> JML.unp(τ) = if $\tau \in \{ \text{float}, \text{double}, \backslash \text{real} \}$ then $\backslash \text{real}$ else $\backslash \text{bigint}$ end JML.bnp(τ_1, τ_2) = if τ_1 or τ_2 is one of $\{ \text{float}, \text{double}, \backslash \text{real} \}$ then $\backslash \text{real}$ else $\backslash \text{bigint}$ end </pre>	<pre> Java.unp(τ) = if $\tau \in \{ \text{byte}, \text{short}, \text{char} \}$ then int else τ end Java.bnp(τ_1, τ_2) = if $\backslash \text{real} \in \{ \tau_1, \tau_2 \}$ then $\backslash \text{real}$ else-if $\backslash \text{bigint} \in \{ \tau_1, \tau_2 \}$ then $\backslash \text{bigint}$ else-if double $\in \{ \tau_1, \tau_2 \}$ then double else-if float $\in \{ \tau_1, \tau_2 \}$ then float else-if long $\in \{ \tau_1, \tau_2 \}$ then long else int end </pre>
---	--

Figure 6. JMLa type conversion functions

4.4 Example

As an example of the application of the elaboration rules we will use a slightly modified version of the specification of `isqrt` given in Figure 1 in which we replace the occurrences of `Math` by `JMLMath`. `JMLMath` is a model class that contains methods like `Math` but that are defined over JMLa arbitrary precision types [Chalin03a]. The result is shown in Figure 7. Notice how the PVS expressions closely resemble their JMLa counterparts. As a partial indication of the suitability of the semantic translation, we have been successful in proving the consistency of the `isqrt` specification.

5 Related work

Several computer languages and tools provide basic language support for arbitrary precision integers including: specification languages, such as B, OBJ, VDM, and Z [Bowen03]; BISLs such as Larch, and Extended ML (EML); Functional languages like ML, Haskell, and Lisp; proof tools (e.g. PVS) and symbolic mathematics systems such as Mathematica and Maple. Basic support for real numbers is most common in general design specification languages and proof tools and less common in other languages. Symbolic mathematics packages often provide arbitrary precision rational numbers.

The LOOP tool, being developed at the University of Nijmegen, translates JML specifications and Java code into the language of PVS [vdBJ01, JP01]. The current LOOP embedding defines a semantics that more closely resembles the proposed semantics for JMLa than that of JML. Unfortunately, the embedding also does not currently support expressions involving *both* arbitrary precision and bounded precision types.

The work described in this extended abstract has been inspired by our previous work on the semantics of LCL, which we also defined by means of an embedding into a logic [CGR96]. Our work on LCL semantics has resumed and we are currently extending and generalizing it [Chalin02]. In another report, we provide an exploration of language design alternatives for the JML support of arbitrary precision numeric types. In this same report, we also offer a more rigorous comparison of the formal semantics of JML, JMLa, Larch/C++ and LCL [Chalin03b].

6 Conclusions and future work

We have illustrated a semantic gap between user expectations of the meaning of expressions over numeric types and the current JML language definition. Due to this gap, several published JML specifications are

```

IMPORTING int, ...

isqrt_requires(y: int): bool = y >= 0;
isqrt_ensures(y, result: int): bool =
  JMLMath.abs(result) <= y AND
  result * result <= y AND
  y < (JMLMath.abs(result) + 1) * (JMLMath.abs(result) + 1);

isqrt_consistent: LEMMA
  FORALL (y:int):
    EXISTS (result:int): isqrt_requires(y) => isqrt_ensures(y,result)

```

Figure 7. PVS definition of `isqrt`

invalid or inconsistent [Chalin03a]—we have presented two such problematic specifications. To better meet user expectations, we have defined a variant of JML called JMLa that has support for primitive arbitrary precision numeric types `\bigint` and `\real`. Furthermore, JMLa defines implicit operand promotion to these arbitrary precision types. Special JMLa operators such as `\nowarn()` allow us to recover Java expression semantics. A preliminary formal semantics of JMLa expressions is given and its application is illustrated by means of a simple example.

Work on the formal semantics of JMLa will be pursued so as to progressively include more language elements. Although PVS has been successfully used to establish proofs of consistency and validity of simple JMLa specifications, larger examples will need to be tackled. We will also continue our analysis of JML as other issues related to bounded vs. unbounded “data types” (such as arrays, sets and sequences) need to be examined. In collaboration with other JML project partners we will be implementing support for `\bigint` and `\real` in ESC/Java and the JML checker. This will allow us, in particular, to support our claim that ESC/Java will be able to detect more errors in JMLa vs. JML specifications. Preliminary implementation of JMLa in the JML checker has already allowed us to detect over two dozen inconsistent or erroneous JML specifications.

7 Acknowledgments

We thank the anonymous referees for their helpful comments on the paper, as well as members of the JML developer community for discussions that have contributed to the improvement of JMLa. Also thanks to Frederic Rioux for his contribution to the implementation of JMLa support in the JML checker.

8 References

- [Bowen03] Jonathan Bowen, *WWW Virtual Library: Formal Methods*, <http://www.afm.sbu.ac.uk>. February 2003.
- [BvdBJ02] C.-B. Breunese, J. van den Berg, and B. Jacobs. Specifying and verifying a decimal representation in Java for smart cards. In H. Kirchner and C. Ringeissen, editors, *AMAST'2002*, LNCS, pp. 304-318. Springer Verlag, 2002. Decimal class specification is available at www.cs.kun.nl/indexes/~ceesb/decimal/Decimal.java.
- [CGR96] Patrice Chalin, Peter Grogono, and T. Radhakrishnan. “Identification of and solutions to shortcomings of LCL, a Larch/C interface specification language”. In Marie-Claude Gaudel and James Woodcock, eds, *FME'96: Industrial Benefit and Advances in Formal Methods*, LNCS 1051, pp. 385-404. Formal Methods Europe, Springer, March 1996.
- [Chalin02] Patrice Chalin. *Formal Semantics of LCL, Revised*. ENCS-CS Technical Note 2002-001.1, Concordia University, 2002.
- [Chalin03a] Patrice Chalin. Improving JML: For a Safer and More Effective Language. In Stefania Gnesi, Keijiro Araki, and Dino Mandrioli (Eds.), *FME 2003, International Symposium of Formal Methods Europe*, Pisa, Italy, Sept. 8-14, 2003, Proceedings (to appear).
- [Chalin03b] Patrice Chalin. *Back to Basics: Language Support and Semantics of Basic Infinite Integer Types in JML and Larch*. ENCS-CS TR 2002-003.4, Concordia University, October 2002; latest revision, July 2003.
- [Flanagan+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In Cindy Norris and James B. Fenwick, editors, *Proceedings of Conference on Programming Language Design and Implementation (PLDI-02)*, volume 37, 5 of ACM SIGPLAN, pages 234-245, June 17-19 2002.
- [GH93] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andres Modet, and Jeannette M. Wing.
- [JP01] Bart Jacobs and Erik Poll. *A Logic for the Java Modeling Language JML*. In: H. Hussmann (ed.), *Fundamental Approaches to Software Engineering (FASE)*, LNCS 2029 pages 284-299. Springer-Verlag 2001.
- [LBR02] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *Preliminary Design of JML: A Behavioral Interface Specification Language for Java*. Computer Science Dept., Iowa State University, TR #98-06t, Dec. 2002.
- [Leavens99] Gary T. Leavens. *Larch/C++ Reference Manual*, Iowa State University, Version 5.41, April 1999.
- [PVS] The PVS Specification and Verification System. <http://pvs.csl.sri.com>.
- [vdBJ01] Joachim van den Berg and Bart Jacobs. *The LOOP compiler for Java and JML*. In: T. Margaria and W. Yi editors, *Tools and Algorithms for the Construction and Analysis of Software (TACAS)*, LNCS 2031, pp. 299-312. Springer, 2001.
- [Wing87] Jeannette M. Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1-24, January 1987.
- [Winskel93] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Computing Series. MIT Press, 1993.

Literals and Logical Variables

$$\frac{}{\rho \vdash c_\tau \longrightarrow c : \tau} \text{ Literal} \quad \frac{\rho(\text{LVAR } \iota) = \tau}{\rho \vdash \iota \longrightarrow \iota : \tau} \text{ Logical Var}$$

Operators

$$\frac{\begin{array}{l} op \in \text{unsafeOps} \wedge \rho(\backslash \text{safeMath}) = \text{true} \\ \rho \vdash e_i \longrightarrow \epsilon_i : \tau_i \quad \text{for } i = 1..k \\ (\tau_1, \dots, \tau_k) \in \text{ArgType}_{op} \quad \text{JML.conv}_{op}(\tau_1, \dots, \tau_k) = (\tau'_1, \dots, \tau'_k) \\ \text{resultType}_{op} = \tau \quad (\tau'_1, \dots, \tau'_k \rightarrow \tau) \in \rho(\text{OP } op).types \end{array}}{\rho \vdash op(e_1, \dots, e_k) \longrightarrow \phi_{op}(\epsilon_1, \dots, \epsilon_k) : \tau} \text{ Op}_{\text{safe}}$$

$$\frac{\begin{array}{l} op \notin \text{unsafeOps} \vee \rho(\backslash \text{safeMath}) = \text{false} \\ \rho \vdash e_i \longrightarrow \epsilon_i : \tau_i \quad \text{for } i = 1..k \\ (\tau_1, \dots, \tau_k) \in \text{ArgType}_{op} \quad \text{Java.conv}_{op}(\tau_1, \dots, \tau_k) = (\tau'_1, \dots, \tau'_k) \\ \text{resultType}_{op} = \tau \quad (\tau'_1, \dots, \tau'_k \rightarrow \tau) \in \rho(\text{OP } op).types \end{array}}{\rho \vdash op(e_1, \dots, e_k) \longrightarrow \phi_{op}(\epsilon_1, \dots, \epsilon_k) : \tau} \text{ Op}$$

Warn/Nowarn Operators

$$\frac{\rho \oplus \{ \backslash \text{safeMath} \mapsto \text{false} \} \vdash e \longrightarrow \epsilon : \tau}{\rho \vdash \backslash \text{nowarn}(e) \longrightarrow \epsilon : \tau} \text{ nowarn} \quad \frac{\rho \oplus \{ \backslash \text{safeMath} \mapsto \text{true} \} \vdash e \longrightarrow \epsilon : \tau}{\rho \vdash \backslash \text{warn}(e) \longrightarrow \epsilon : \tau} \text{ warn}$$

$$\frac{\begin{array}{l} e'_i = \backslash \text{warn}(e_i) \quad \text{for } i = 1..k \\ \rho \vdash \backslash \text{nowarn}(op(e'_1, \dots, e'_k)) \longrightarrow \epsilon : \tau \end{array}}{\rho \vdash \backslash \text{nowarn_op}(op(e_1, \dots, e_k)) \longrightarrow \epsilon : \tau} \text{ nowarn_op} \quad \frac{\begin{array}{l} e'_i = \backslash \text{nowarn}(e_i) \quad \text{for } i = 1..k \\ \rho \vdash \backslash \text{warn}(op(e'_1, \dots, e'_k)) \longrightarrow \epsilon : \tau \end{array}}{\rho \vdash \backslash \text{warn_op}(op(e_1, \dots, e_k)) \longrightarrow \epsilon : \tau} \text{ warn_op}$$

Casts

$$\frac{\rho \vdash e \longrightarrow \epsilon : \tau' \quad \tau' \leq \tau}{\rho \vdash (\tau)e \longrightarrow \epsilon : \tau} \text{ Widen} \quad \frac{\rho \vdash e \longrightarrow \epsilon : \tau' \quad \tau' > \tau}{\rho \vdash (\tau)e \longrightarrow \text{narrow}(\epsilon) : \tau} \text{ Narrow}$$

Old & Quantifier Expressions

$$\frac{\rho \oplus \{ \backslash \text{state} \mapsto \text{'pre'} \} \vdash e \longrightarrow \epsilon : \tau}{\rho \vdash \backslash \text{old}(e) \longrightarrow \epsilon : \tau} \text{ old} \quad \frac{\begin{array}{l} \rho \oplus \{ \text{LVAR } \iota \mapsto \tau \} \vdash e \longrightarrow \epsilon : \text{boolean} \\ \tau \in \text{PrimitiveNumeric} \end{array}}{\rho \vdash (q \tau \iota e) \longrightarrow (q' (\iota : \tau) : \epsilon) : \text{boolean}} \text{ Quant}$$

Static Field/Method Access

$$\frac{\begin{array}{l} (\text{CLASS } \iota') \in \text{dom } \rho \\ \text{lookup}_{tp}(\rho, \iota', \iota) = \tau \\ \sigma = \rho(\backslash \text{state}) \end{array}}{\rho \vdash \iota'.\iota \longrightarrow \text{val}(\sigma, \iota'.\iota) : \tau} \text{ Static Field} \quad \frac{\begin{array}{l} \rho \vdash e_i \longrightarrow \epsilon_i : \tau_i \quad \text{for } i = 1..k \\ (\text{CLASS } \iota') \in \rho \\ \text{lookup}_{tp}(\rho, \iota', \iota) = [\tau_1, \dots, \tau_k \rightarrow \tau'] \end{array}}{\rho \vdash \iota'.\iota(e_1, \dots, e_k) \longrightarrow \iota'.\iota(\epsilon_1, \dots, \epsilon_k) : \tau'} \text{ Static Method}$$

Figure 8. JMLa expression semantics, selected rules

Verifying JML specifications with model fields

Cees-Bart Breunesse* and Erik Poll

Department of Computer Science, University of Nijmegen

Abstract. The specification language JML (Java Modeling Language) provides *model fields* as a means to abstract away from implementation details in specifications. This paper investigates how specifications with model fields can be translated to proof obligations. In order to do this, the semantics of model fields has to be made precise.

1 Introduction

The Java specification language JML (Java Modeling Language) [LBR01] provides *model fields* to allow specifications to abstract away from implementation details [CLSE03]. Model fields in JML play the role that abstract values play for traditional abstract data types. The idea is that clients of a class can use an abstract specification in terms of model fields, and that only when implementing the class we have to know how these abstract model fields are related to the actual concrete representation.

The ability to handle model fields is vital if our efforts at program verification with the LOOP tool [BJ01], or any other verification technology, are to scale up to bigger programs. This paper investigates how specifications with model fields can be translated to proof obligations as Hoare triples, or rather n-triples in the style of [JP01]. We only look at this issue from the point of view of someone implementing a class whose specification relies on model fields. The translation of specifications into these proof obligations is being incorporated in the LOOP tool, but we describe the translation at a relatively high level of abstraction here, so that it should be useful for other groups working on verification tools. More generally, it can be seen as an attempt at precisely defining the semantics of JML. To better understand model fields, we investigate several syntactic (de)sugarings of JML specifications with model fields. This paper does not try to give a definite answer on how to handle model fields. It is an attempt to clarify some issues, and to show possible solutions. Further investigations and case-studies must show which approach must be used in what setting.

The outline of the rest of the paper is as follows. Section 2 gives a brief introduction to the use of model fields in JML specifications. There are two ways in which an implementation can declare its concrete representation of a model field in JML. The first and simpler way is to give an explicit definition of this representation, by means of a *representation function*, using the operator `<-`. This is discussed in Section 3. The second, more general, way is to give a *representation relation* between the model field and its representation, using the keyword `\such_that`. This is discussed in Section 4. Dealing with these `\such_that` clauses poses the main challenge.

```

interface MyIntInterface {
    /*@ model int myint;
        invariant 0 <= myint && myint < 512 @*/

    /*@ requires    true;
        assignable myint;
        ensures     myint == (\old(myint) + 1) % 512; @*/
    void succ();
}

```

Fig. 1. Java/JML Interface: `MyIntInterface`

2 JML specifications with model fields

Our (extremely simple) running example of a JML specification with a model field is given in Fig. 1. It declares a model field `myint` that represents an integer value between 0 and 512. Such a model field is an imaginary field that can only be used for specification purposes. For instance, in Fig. 1, the model field `myint` is used in the specification of the method `succ`. This specification consists of several clauses:

- the **requires** clause specifies the pre-condition,
- the **assignable** clause lists fields or model fields which may be altered (in the case of a model field we means its dependees),
- the **ensures** clause specifies the post-condition that must be satisfied when `succ` terminates normally,
- the **invariant** clause specifies a class invariant, which is implicitly included in all pre- and postconditions.

For a client – i.e. someone using objects of type `MyIntInterface` – dealing with such a model field is quite different than for someone who is implementing a class that implements `MyIntInterface`:

- Essentially, clients can treat a model field as a Java field, and forget about the fact that it's not a real field, but that there is some representation hiding behind a model field. Unfortunately, this is only true if the representation is properly encapsulated, and if there are no unexpected ways in which one might alter the representation of the model field `myint`. Ensuring such proper encapsulation is an important and active research topic. A solution has been proposed for JML, using so-called universes, in [MPHL03]. We will completely ignore this issue in this paper, and we can, because we only look at model fields from the point of view of someone implementing a class that has model fields.
- Implementors will have to specify how the abstract model field `myint` is represented in the concrete implementation. In JML, this is done with so-called **represents** and **depends** clauses. These determine what concrete proof obligations follow from the abstract specification. This is the topic of this paper.

There are two kinds of represents clause:

- The first (using the `<-` notation) gives an explicit definition of the model field in terms of the real fields, by means of a representation *function*. For an example, see Fig. 2. This example, and the way to handle it, is discussed in Section 3. Dealing with such a represents clause is easy: we can basically treat the model field as a macro that is expanded to get the concrete proof obligations.

* Supported by the EU IST project VerifiCard (IST-2000-26238)

```

class MyIntImpl1 implements MyIntInterface {
    /*@ represents myint <- (f ? 256 : 0) + (b & 0xFF);
        depends    myint <- b,f; @*/

    byte b;
    boolean f;

    void succ() {
        f = (b == -2) ^ f;
        b = (byte)(b + 1);
    }
}

```

Fig. 2. An implementation using `<-`: `MyIntImpl1`

- The second (using the `\such_that` notation) more general kind only specifies a representation *relation* between the model field and the real fields. For an example, see Fig. 3. Dealing with these representation relations is more complicated, and is the main topic of this paper. This is discussed in Section 4.

3 Model fields with representation functions (using `<-`)

In JML an explicit definition of the model field in terms of the real fields can be given using a `represents` clause of the following form

```
/*@ represents myint <- ...
```

Here the expression to the right of the `<-`, in this case an integer expression, gives the *representation function* for the model field.

For example, in Fig. 2 a boolean and a byte are used to implement `myint`. Together, a boolean and a byte can represent a 9 bit numerical value, i.e. an integer between 0 and 512. The boolean `f` in Fig. 2 serves as the most significant bit. The `represents` clause in Fig. 2 gives an explicit definition of the model field `myint` in terms of the concrete representation in the field `b` and `f`.¹

3.1 Syntactic desugaring: model fields as macros

To verify that the concrete implementation `MyIntImpl1` given in Fig. 2 meets the abstract specification `MyIntInterface` given in Fig. 1 is relatively straightforward. Basically, we can just treat the model field `myint` in the abstract specification `MyIntInterface` as a macro, and expand it to obtain the concrete proof obligation for `MyIntImpl1`. More specifically:

- we replace all occurrences of `myint` in assignable clauses by the `depends` clause specified in `MyIntImpl1`, i.e. by “`b,f`”.
- we replace all other occurrences of `myint` by the `represents` clause specified in `MyIntImpl1`, i.e. by “`(f ? 256 : 0) + (b & 0xFF)`”.

¹ This involves some Java technicalities: the Java expression `b & 0xFF` is the unsigned interpretation of the byte `b`, i.e. a value between 0 and 255.

```

class MyIntImpl2 implements MyIntInterface {
  /*@ represents myint \such_that (!f ==> myint == (b & 0xFF)) &&
                                     ( f ==> myint == (b & 0xFF) + 256);

    depends myint <- b,f; @*/

  byte b;
  boolean f;

  void succ() {
    ...
  }
}

```

Fig. 3. An implementation using `\such_that`: `MyIntImpl2`

Note that we then end up with a specification which no longer refers to a model field. We can deal with such specification already with the LOOP tool, so we can verify that `MyIntImpl1` meets this specification in the standard way.

This macro expansion is not done at the syntactical level by the LOOP tool, but is incorporated in the translation of JML to PVS. Basically, a model field is treated as a Java field, but instead of looking up its value from the heap, its value is defined using the representation function.

4 Model fields with representation relations (using `\such_that`)

Figure 3 shows class `MyIntImpl2`, which has the same implementation as class `MyIntImpl1`, but a different represents clause. The represents clause here uses a `\such_that` clause to specify a relation between the model field `x` and the real fields `f` and `b`. In this particular example, the relation is a total, one-to-one relation: for any given values of `b` and `f` there exists exactly one value of `myint` in the representation relation. In fact, the representation relation is simply the representation function specified in Figure 2, written in a different way.

However, in general a representation relation need not be a function. For example, the representation relation in Figure 4 is a partial relation, and it is many-to-one: given values of the concrete fields `j` and `k` there can be many values, or none, for the model field `x` that are in the representation relation.

Dealing with such representation relations, that are not total and one-to-one, is more difficult than dealing with functionals ones. We therefore distinguish three kinds of rep clauses:

1. “rep functions”, written using the `<-` notation,
2. “functional rep relations”, written using `\such_that`, but which specify a total, one-to-one function.
3. “proper rep relations”, written using the `\such_that`, that are proper relations, i.e. not functions.

A reason for using the second kind of rep clause rather than the first can be convenience; some people may find it easier to read the rep clause in Fig. 3 than the one in Fig. 2.

```

class A {
  int j,k;
  //@ model int x;
  //@ represents x \such_that (j <= x && x <= k);

  /*@ requires true;
    ensures false; */
  void m() {
    j = 1;
    k = 0;
    // there is no x such that j <= x <= k !
  }

  void n() {
    j = 10;
    k = 12;
    //@ assert x == 10;
  }
}

```

Fig. 4. Example class A

For the remainder of this section, suppose we have a model field and represents clause as follows:

```

//@ model t x;
//@ represents x \such_that R(x,  $\bar{q}$ );

```

where

- t is a type,
- x is a model field,
- $\bar{q}$ is a vector of fields which make up the concrete representation of x ,
- R is the rep relation between x and $\bar{q}$.

In class A of Fig. 4, the vector $\bar{q}$ contains j and k .

4.1 Axiomatic approach

The approach used in [Lei00] is to consider the representation relation R as an axiom. This means one simply assumes that, at any point in the program, the model field x has some value satisfying the representation relation R . In other words, we treat $R(x, \bar{q})$ as an axiom that holds at every program point:

Approach 1 *At any point in the program we assume that the model field x has some value such that $R(x, \bar{q})$ holds.*

This approach is sound for rep functions and functional rep relations, but unsound for some proper rep relations. Consider for example the class A in Fig. 4. The, clearly incorrect, specification for method m can be proved correct using approach 1: if we assume that $(j \leq x \ \&\& \ x \leq k)$ holds at the end of the method $m()$, then we have a contradiction, and we can deduce the postcondition **false**.

Clearly, the problem with using R as an axiom arises in program states for which there is no x that satisfies R , i.e. if our representation relation is not total. Because [Lei00] considers only rep functions or functional rep relations, the axiomatic approach is sound in that setting.

4.2 Desugaring approach

Proper rep relations are tricky. There may be program points in which there is no value for the model field $\mathbf{x}$ that satisfies R . We therefore cannot assume that R holds everywhere, as the example above illustrates. When the model field $\mathbf{x}$ is mentioned in say a postcondition, we must make sure that there actually exists a value for x that satisfies R .

We can try to formalise this idea by desugaring every JML predicate (i.e. every JML boolean expression in a **requires**, **ensures**, **assert** or **invariant** clause) into an existential quantification which no longer has free occurrences of a model field $\mathbf{x}$. Such syntactic desugarings are a convenient way to explain many details about the semantics of JML, e.g. see [RL00].

Approach 2 *Any JML predicate $A(\mathbf{x})$ that possibly refers to the model field $\mathbf{x}$ is desugared to the JML predicate*

$$(\exists x; R(\mathbf{x}, \bar{q}) \ \&\& \ A(\mathbf{x}) \)$$

where R is the representation relation.

Again, let us consider class **A** in Fig. 4 to observe whether approach 2 makes sense. We no longer have the problem with the specification for method **m** that occurred in the axiomatic approach, as clearly we cannot establish the postcondition

$$(\exists \text{int } x; j \leq x \ \&\& \ x \leq k \ \&\& \ \text{false})$$

for method **m()**.

However, now we have a problem with method **n()**. Applying the desugaring given in approach 2 to the assert in method **n** results in the desugared assert:

```
//@ assert (\exists int x; j <= x && x <= k && x == 10);
```

This assertion is clearly true. However, the representation relation also allows 11 or 12 as possible values for the model field $\mathbf{x}$, if $j == 10$ and $k == 12$. So the assertion that $x == 10$ does not necessarily hold at the end of **n()**, and should not be provable.

This illustrates that approach 2 is not what we want. The problem is caused by the fact that the representation relation is many-to-one. A more complicated desugaring, which we argue is sound, is to say that an assertion of the form

```
//@ assert A(x);
```

means that A holds for *every* possible value of $\mathbf{x}$ that satisfies the representation relation R , instead of just one, which is expressed approached 2. Moreover, there should be at *at least one* possible value for $\mathbf{x}$ satisfying R . This leads to our second desugaring approach:

Approach 3 Any JML predicate $A(x)$ that possibly refers to the model field x is desugared to the JML predicate

$$(\forall t \ x; R(x, \bar{q}) \implies A(x)) \ \&\& \ (\exists t \ x; R(x, \bar{q}))$$

where R is the representation relation.

Taking approach 3 the assert in method `n` is desugared to

```
//@ assert (\forallall int x; (j <= x && x <= k) => x == 10) &&
//@      (\exists int x; (j <= x && x <= k));
```

for which the first part of the conjunction resolves to false, which is what we would expect. The strongest condition that we would expect to hold after execution of `n` is

```
//@ assert x == 10 || x == 11 || x == 12;
```

which covers the full range of possible values of x . Using approach 3, this is indeed the strongest postcondition that can be established.

Although desugaring approach 3 is what we need to tackle representation relations, the existential quantifications it involves can be quite a burden during proofs: during the verification of a method, at many times will witnesses for these existential quantifications have to be constructed by hand. The existential quantifier pops up every time a JML predicate referencing x is encountered. For fully annotated programs, these occurrences can be numerous.

To avoid giving a witness every time a model field is encountered, the person writing the specs might check whether the represents clause is a functional represents relation, which can directly be rewritten to a rep function using `<=`. If we are actually confronted with a proper represents relation, there is another way to avoid the problem with existential quantifiers, which we discuss in the next section.

4.3 Model fields as methods

An alternative desugaring of JML specifications that use model fields to JML specifications that does not use a model field, is to replace a model field by a method whose postcondition corresponds to the representation relation. (This method should maybe be a specification-only method, or, in JML-speak, a model method, but we ignore that issue here.)

For example, the model field `myint` used in Figure 3 could be desugared as shown in Figure 4.3. Note that the postcondition of the method `myint()` in Figure 4.3 is exactly the representation relation for the model field `myint` given in Figure 3.

The need for the keywords `pure` and `helper` is rather technical, and probably best ignored by people not familiar with or interested in the obscure details of JML². It is of course simple to give an implementation of `myint()`, but we do not really have to do this, if we are satisfied with reasoning about `myint()` on the basis of its specification.

² The keyword `pure` means that the method `myint()` does not have any side-effects; this is required in order to be allowed to use the method in specifications, e.g. the postcondition of `succ()` and the invariant. The keyword `helper` means that the invariant is *not* implicitly included in the pre- and postcondition of `myint()`.

```

class MyIntImpl2 implements MyIntInterface {

    byte b;
    boolean f;

    /*@ requires true;
        ensures (!f ==> \result == (b & 0xFF)) &&
              ( f ==> \result == (b & 0xFF) + 256); @*/
    /*@ pure helper @*/ int myint() {
        ...
    }

    /*@ invariant 0 <= myint() && myint() < 512;

    /*@ requires true;
        assignable b,f;
        ensures myint() == (\old(myint()) + 1) % 512;
        @*/
    void succ() {
        ...
    }
}

```

Fig. 5. Replacing model field by a method

In general, replacing a model field by a method can be more complicated than in the example above. If we have a partial representation relation, as was the case in the class A in Fig. 4, things are a bit more complicated. A first attempt at replacing the model field `x` by a method for class A in Fig. 4 might be:

```

class A{
    int j,k;

    /*@ requires true;
        ensures j <= \result && \result <= k @*/
    /*@ pure helper @*/ int x() {
        ...
    }

    /*@ requires true;
        ensures false; @*/
    void m() {
        j = 1;
        k = 0;
        // there is no x such that j <= x <= k !
    }
    ...
}

```

However, here we run into the same problem as for approach 2, namely that we can prove the correctness of the – clearly inconsistent – spec for `m()`, if we assume the specification of `myint()` is correct. Again, the problem is that in the post-state of `m()` there exists no value for `x()` satisfying its spec.

The solution to this is that we should specify a suitable precondition for `x()` which ensures that its postcondition is satisfiable, e.g.

```
/*@ requires j <= k;
    ensures j <= \result && \result <= k @*/
/*@ pure helper @*/ int x()
```

This specification for `x()` is satisfiable, something which could be established by proving

$$(j \leq k) \implies (\exists \text{int } x; j \leq x \leq k)$$

or by providing an implementation for `x()` and proving that it meets the specification.

Let us make the approach above a bit more precise:

Approach 4 *We introduce a method `x()` to replace a model field `x`, and*

- *replace every occurrence of the model field `x` in a JML predicate by a call to the method `x()`,*
- *replace the declaration of model field `x` by a pure helper method `x()`, with a suitably chosen precondition P , and as a postcondition the representation relation, with `\result` substituted for `x`, i.e.*

```
/*@ requires P;
    ensures R(\result, \bar{q}) */
private model helper int x()
```

for which one proves

$$P \implies (\exists t; R(x, \bar{q}))$$

possibly by giving an implementation for `x()` and proving that it meets its specification.

The main advantage of using approach 4 instead of approach 3 is that, provided P can be kept simple, it is less work to prove P than to construct a witness `x` that satisfies $R(x, \bar{q})$, at every program point where the model field is used in a JML predicate.

5 Conclusions and Future Work

In this paper we looked only at model fields from the perspective of someone implementing a class that has model fields, not from the perspective of a client using such a class. This implementors' perspective was definitely more complex than we first anticipated, due to the possibility of representation *relations*, which may be partial or many-to-one, rather than using a representation *functions*.

Handling representation functions is relatively straightforward, but representation relations, when they are not total functions in disguise, are a big burden when it comes to verifying programs, as illustrated in Section 4.2. The alternative approach of looking at model fields as methods, discussed in Section 4.3, can save a lot of effort in verifying programs, at the cost of some extra work in specifying, namely coming up with a suitable precondition for the method replacing the model field.

Note that the approach discussed in Section 4.3, to introduce methods for model fields, can also be useful when doing runtime assertion checking. The JML runtime assertion checker [Che03,CL02] can perform runtime checks for assertions that involve a model field, but only if a representation function is specified using `<-`, not if a `\such_that` clause is given. For a model field `x` with a very complicated, but functional, `rep` relation, it usually is possible to define a method `x()` that computes this function and then replace the original `\such_that` clause by “`arepresent x <- x();`”. Doing this would allow more assertions to be checked at runtime.

So far, we have only investigated relatively simple examples using model fields. For more complicated examples, that make use of the JML model classes which represent mathematical concepts such as sets and relations, we would of course like to express the proof obligations not in terms of Java implementations of sets and relations, but directly in terms of the underlying mathematical notions. (In our case, because we use PVS to prove our proof obligations, we would like to use PVS sets and relations.) This is still left as future work.

Acknowledgements

We would like to thank the anonymous referees for their comments, which were a great help in improving the paper.

References

- [BJ01] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *TACAS’01*, volume 2031 of *LNCS*, pages 299–312. Springer, Berlin, 2001.
- [Che03] Yoonsik Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Iowa State University, 2003. Available as Technical Report TR 03-09, Department of Computer Science, Iowa State University.
- [CL02] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *the International Conference on Software Engineering Research and Practice (SERP ’02)*, pages 322–328. CSREA Press, June 2002.
- [CLSE03] Yoonsik Cheon, Gary T. Leavens, Murali Sitaraman, and Stephen Edwards. Model variables: Cleanly supporting abstraction in design by contract. Technical Report 03-10, Department of Computer Science, Iowa State University, April 2003. Available from archives.cs.iastate.edu.
- [JP01] Bart Jacobs and Erik Poll. A logic for the Java Modeling Language JML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE)*, volume 2029 of *LNCS*, pages 284–299. Springer, Berlin, 2001.
- [LBR01] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06p, Iowa State University, Department of Computer Science, August 2001. See www.jmlspecs.org.
- [Lei00] Greg Leino, K. Rustan M. and Nelson. Data abstraction and information hiding. Technical Report SRC 160, Compaq Systems Research Center, 2000.
- [MPHL03] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular specification of frame properties in JML. *Concurrency and Computation: Practice and Experience*, 15(2):117–154, 2003.
- [RL00] Arun D. Raghavan and Gary T. Leavens. Desugaring JML method specifications. Technical Report 00-03a, Iowa State University, Department of Computer Science, July 2000.

Verification of Object-Oriented Programs with Invariants

Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte

Microsoft Research, Redmond, WA, USA

{mbarnett,rdeline,maf,leino,schulte}@microsoft.com

Manuscript KRML 122a, 2 May 2003.

Abstract. This extended abstract outlines a system for the modular verification of an object-oriented programming language. While simplified, the language has object and array references, single-inheritance subclassing, and single-dispatch methods. Programs are verified against specifications consisting of *pre- and post-conditions* for methods, and *object invariants* stating the consistency of data. The meaning of a program is given by its translation into *verification conditions*: logical formulas that are valid if and only if the program is consistent with its specification. The technique is *sound* and is expressive enough to allow many interesting object-oriented programs to be specified and verified. The key idea is to keep track of, for each object, which invariants the object satisfies and whether or not the object can be mutated.

0 Introduction

Writing correct programs is difficult, and useful modern programming-language features like information hiding and object orientation provide several challenges in reasoning about programs. In this extended abstract, we consider a programming methodology based on method specifications and object invariants and define the correctness criteria for programs specified in that way. The key idea is to keep track of which object invariants each object satisfies at each program point (which we do by introducing for each object a special field *inv*) and whether or not the object is in a state where it can be mutated (which we do by introducing for each object a special field *exposed*).

The main characteristics of this work are:

- *Ease of use.* Our verification system needs only a relatively small repertoire of programming concepts. This makes it simpler than previous verification approaches. For instance, alias confinement and readonly modifiers are not first-class concepts in our methodology. Also, our technique does not require as much foresight on behalf of the program designers as does the *valid/state paradigm* in ESC/Modula-3 [5, 10]. Perhaps surprisingly, our technique does not even use features of abstraction to specify which variables a method is allowed to modify (like abstract variables and abstraction dependencies [10] or data groups [11]).

- *Simple foundation.* The meaning of programs is given by a translation from programs to *verification conditions*, thus following the approach ESC/Modula-3. Verification conditions are expressed in first-order predicate calculus. Thus, we do not need any special logics, like separation logics (*e.g.*, [1]), or coalgebras as used in the LOOP project [9], to reason about pointer structures and updates. This makes our approach more amenable to further studies on heap structures and confinement.
- *Modularity.* The proposed technique is modular in that one can verify software units (say, methods, classes, or small sets of classes) separately and independently from the rest of the program. In contrast to whole-program analysis, this supports scalability and fast response times.
- *Soundness.* We claim our technique to be *modularly sound*: if the technique detects no violations in the independently verified software units, then there are no violations in the program as a whole. Modular soundness was a goal in ESC/Modula-3, but it was achieved only for the most basic forms of its specification language. Using a different encoding of similar specifications, Müller and Poetzsch-Heffter’s technique [14, 13] achieves modular soundness.

Other related static-checking systems are Vault [3, 7] and Fugue [4], which incorporate some modern alias confinement techniques like *capabilities* [17]. Our approach has a similar methodology, but aims at analyzing the programs at a more detailed level.

1 Methodology

As a first part of our programming methodology, each object can either be *exposed* or not. A program can only write the fields on an object when it is exposed. When an object is not exposed, it exists as an integral *component* of some other object and is said to be *owned* by that other object.

Although a program cannot write the fields of an owned object, the program can read its fields. The idea is that an owned object is encapsulated; other objects should not care or depend upon its details and should not be surprised if its details change as the result of any method call.

In an object-oriented language, an object can have multiple declared types due to inheritance. We allow the programmer to specify an invariant in each class declaration, which means that an object’s complete invariant consists of the invariants of all the inherited classes. As a second part of our methodology, we allow a method to specify which subset of an object’s total invariant holds in the method’s pre- and postcondition. For simplicity, this extended abstract considers an object-oriented language with single inheritance, like C# or Java, but without the **interface** feature of these languages. As part of our methodology, we restrict the invariant subset to be a *prefix* of an object’s invariants, from those declared in the root class **object** toward those declared in the allocated type of the object. We represent this invariant prefix using a single type, namely, the most derived type whose invariant holds.

The two parts of the methodology are related by the following two properties:

0. If, for an object, the invariants declared in a class T are known to hold, then all of the *components* of the object declared in class T are *owned* by the object.
1. If an object is owned, then all its object invariants are known to hold.

We now make these notions more precise. Every object is extended by two instance fields, *exposed* and *inv*. The field *exposed* holds a boolean that represents whether the object's fields can be written. The field *inv* holds a type that records the most derived class whose declared invariants are known to hold for the object. If no invariants are known to hold for the object, *inv* holds the special value $\perp$.

The fields *exposed* and *inv* are special in that they can be mentioned in method specifications but not directly read or written in method bodies. To update these special fields, we introduce two statements **pack** and **unpack**, which delineate the scope in which invariants hold. The statement **unpack** *o* **from** *T* changes *o.inv* from *T* to the superclass of *T*. Statements after **unpack** can update fields and sub-components declared in *T* and violate *T*'s declared invariant. The statement **pack** *o* **as** *T* checks that *T*'s declared invariant holds and changes *o.inv* from the superclass of *T* back to *T*. This description should be sufficient to study a small example, in which the meaning of **pack** and **unpack** is explained in greater detail.

2 Example

We consider a class *Bag*, whose instances represent multisets (bags) of integers. The implementation has two fields: an integer *n* that stores the cardinality of the bag, and an array *a* whose first *n* elements are the elements of the bag. The class is declared (in our simple object-oriented language) as follows:

```
class Bag extends object {
  field a: array of int ;
  field n: int ;
  invariant self.a ≠ null ∧ 0 ≤ self.n ∧ self.n ≤ length(self.a) ;
  component self.a ;
  :
}
```

We write **self** for the implicit receiver parameter. Our arrays are like those in C# and Java: they are references to sequences of elements. The **component** declaration specifies a set of references that, in the steady state of a bag object, are owned by the bag (the reason for this particular declaration is explained below). The **invariant** declaration specifies a part of the internal consistency of the *Bag* representation.

Let's consider an instance method that inspects a bag without modifying it:

```
method Contains(x: int) returns (b: bool)
  requires self.inv = Bag
  modifies
  ensures true
{
  var i: int in
    b := false ; i := 0 ;
    while i < self.n do
      b := b ∨ (self.a[i] = x) ;
      i := i + 1
    end
  end
}
```


The method requires the bag's invariant to hold on entry, claims that nothing is modified, and allows an arbitrary return value (reflecting the fact that we are aiming only for a partial specification, like in ESC [5], not a full functional-correctness specification). According to our methodology, the condition $\text{self.inv} = \text{Bag}$ implies that the invariant declared in *Bag* (and also any invariant declared in the superclass **object**) holds. The body of the method can therefore rely on the condition $\text{self.n} \leq \text{length}(\text{self.a})$, which together with the guard $i < \text{self.n}$ ensures that the array reference $\text{self.a}[i]$ does not overstep the array index bounds.

In our simple language, the constructors in C_‡ and Java are broken up into an allocation operation (**new**, which produces a new exposed object) and an explicit call to an initialization routine. The latter is just an *object procedure* (which in C_‡ would be called a *non-virtual method*). The class *Bag* provides the following initializing routine, which establishes *Bag*'s invariant:

```

proc FromArrayElements(e: array of int)
  requires  $\text{self.exposed} \wedge \text{self.inv} = \perp \wedge e \neq \text{null}$ 
  modifies  $\text{self}\{Bag\}, \text{self.inv}$ 
  ensures  $\text{self.inv} = Bag$ 
{
   $\text{self.ObjectInit}()$  ;
   $\text{self.n} := \text{length}(e)$  ;
   $\text{self.a} := \text{new int}[\text{self.n}]$  ;
   $\text{self.a.CopyFrom}(e, 0, \text{self.n})$  ;
  pack  $\text{self}$  as Bag
}

```

The precondition essentially says that **self** is a just-allocated object. The term $\text{self}\{Bag\}$ in the modifies clause grants the license to modify all fields declared in *Bag* or in its superclasses (not counting the special fields *exposed* or *inv*). The modifies clause and postcondition also state that the object procedure returns after having established all of **self**'s invariants down to *Bag*.

The body of the initialization routine starts by calling an initialization routine of the superclass (here called *ObjectInit*). The **pack** statement at the end of the body has the effect of the following pseudo-code, here described more generally for **pack** *o* **as** *T*:

```

assert  $o \neq \text{null} \wedge o.exposed \wedge o.inv = \text{super}(T)$  ;
assert  $\text{Inv}(T, o)$  ;
foreach  $p \in \text{Comp}(T, o)$  do
  if  $p \neq \text{null}$  then
    assert  $p.exposed \wedge p.inv = \text{type}(p)$  ;
     $p.exposed := \text{false}$ 
  end
end
 $o.inv := T$ 

```

where the **assert** statements indicate conditions that are checked, $\text{super}(T)$ denotes the immediate superclass of *T* (or $\perp$ if *T* is **object**), and $\text{Inv}(T, o)$ and $\text{Comp}(T, o)$ denote the invariants and components, respectively, declared in class *T* for the object *o*, and function *type* gives the dynamic type of an object.

Next, let's consider a method that updates a bag:

```

method Add(x: int)
  requires self.exposed ∧ self.inv = Bag
  modifies self.n, self.a
  ensures true
{
  unpack self from Bag ;
  if self.n = length(self.a) then
    var t: array of int in
      t := new int[2 · self.n + 1] ;
      t.CopyFrom(self.a, 0, self.n) ;
      self.a := t
    end
  end ;
  self.a[self.n] := x ;
  self.n := self.n + 1 ;
  pack self as Bag
}

```

Typical of an update method, this method body is bracketed by **unpack** and **pack**, where a statement **unpack *o* from *T*** has the effect of the following pseudo-code:

```

assert o ≠ null ∧ o.exposed ∧ o.inv = T ;
o.inv := super(T) ;
foreach p ∈ Comp(T, o) do
  if p ≠ null then p.exposed := true end
end

```

Note that method *Add* has the precondition **self.exposed**, which allows its implementation to perform **unpack** and **pack** operations on **self** and to modify the fields of **self**. The implementation also modifies the state of **self.a** (namely, its array elements). This modification requires **self.a.exposed**, but it would not be good information-hiding practice explicitly to give **self.a.exposed** in the precondition of *Add*. It is for this reason that **self.a** is listed in a **component** declaration, for then the **unpack** statement establishes **self.a.exposed**. Being a component also means that, in the steady state of the bag **self**, the array **self.a** is owned by **self**. In other words, the array underlying a bag is not used as a component by any other object.

Our methodology and specifications are powerful enough to indicate whether or not a parameter is *captured* by a method, that is, if the method takes over the ownership of the parameter. This is illustrated by the following initialization routine:

```

proc FromArray(e: array of int)
  requires self.exposed ∧ self.inv = ⊥ ∧
    e ≠ null ∧ e.exposed ∧ e.inv = type(e)
  modifies self.{Bag}, self.inv, e.exposed
  ensures self.inv = Bag
{
  self.ObjectInit() ;
  self.n := length(e) ;
  self.a := e ;
  pack self as Bag
}

```

The **pack** statement at the end of the routine body requires all components, which includes `self.a` to be exposed, so that they can be changed into being owned by the bag `self`. The precondition of the object procedure facilitates this, and the modifies clause gives the license to return from the routine having changed the value of `e.exposed`. The modifies clause tells callers that, although they can retain the reference `e`, they are no longer able to own `e`. Note also that our methodology detects the error of trying to implement routine *FromArrayElements* by the body of *FromArray* (because the precondition of **pack** would fail).

Of course, this example gives only the gist of the specification methodology. We do not yet have enough experience to say how powerful our approach is, but look forward to a fruitful discussion at the workshop.

3 Technical Approach

To define the semantics of our object-oriented language in the full paper, we translate it into a more primitive language (cf. [12]), which has a semantics defined by *weakest preconditions* [6, 16]. An object-oriented program is deemed correct if and only if the resulting primitive program is correct. In this section, we make a few remarks about the full system.

We require a kind of *prefix closure* property of declared components: for any subexpression `p.f` occurring in $Comp(T, \text{self})$ or $Inv(T, \text{self})$, either `p` is the literal `self` or `p` occurs in $Comp(T, \text{self})$. Due to space limitations in this extended abstract, we omit here our treatment of array elements in **invariant** and **component**.

Our programming language guarantees the following two *system invariants* (stated informally in Section 1):

0. $(\forall o, T, S \bullet o.inv = T \wedge T <: S \Rightarrow Inv(S, o) \wedge (\forall c \bullet c \in Comp(S, o) \wedge c \neq \text{null} \Rightarrow \neg c.exposed))$
1. $(\forall o \bullet \neg o.exposed \Rightarrow o.inv = type(o))$

where `o` ranges over non-null allocated objects, `T` and `S` range over object types (which does not include $\perp$), and `c` ranges over expressions. These system invariants always hold, because:

- any newly allocated object `o` satisfies $o.exposed \wedge o.inv = \perp$,
- the only statements that directly modify *inv* and *exposed* are **pack** and **unpack**, which maintain the system invariant,
- the value of $Inv(T, o)$ can be changed only by changing the value of some subexpression `p.f` in $Inv(T, o)$; note that if `p` is `o`, then `p.f` can be changed only if `o.exposed`; and note that if `p` is not `o`, then `p.f` is in $Comp(T, o)$, so `p.exposed` only if `o.inv` is a proper superclass of `T`.

The notion of objects being, at any moment, either exposed or owned gives us a way to interpret modifies clauses without the need for other abstraction features. We interpret a modifies clause *M* of any method as giving the license to modify the heap, subject to the following constraint, relating the pre-state and post-state of the method:

$$(\forall o, f \bullet o.f = \text{old}(o.f) \vee o \in \text{old}(M) \vee \neg \text{old}(o.exposed))$$

where o ranges over non-null objects allocated in the pre-state and f ranges over field names. The first two disjuncts are standard, and say that either $o.f$ is not changed or $o.f$ is allowed to be modified according to the modifies clause (as interpreted in the pre-state). The third disjunct is new, and says that fields of owned objects may change during the method call.

4 Discussion

We are embarking on a project to write an invariant checker for the .NET platform and are considering the presented methodology as the foundation of the checker. Our full paper describes more details, but more needs to be done:

- *Expressive power of the language.* So far, we have only made a rudimentary investigation to see what kind of object-oriented programs we can verify. We expect our invariant declarations to be as easy to use as those in ESC/Java [8], but here one needs to explicitly give preconditions about invariants. Being explicit about that seems to help our system (unlike ESC/Java) to be sound. Constrained by soundness, our system is still flexible, in part due to the fact that one can express that only certain invariants hold (a feature not present in ESC/Java). Also, as we formulated it, packing succeeds only if the component graph (meaning the graph with objects as vertices and declared components as edges) from the object being packed is a tree. However, we think that useful relaxations of this rule exist.
- *Interdependence of effects and uniqueness.* Our system allows the reading of arbitrary objects, but allows updates only to exposed objects. One obtains this *exposed* bit from preconditions, newly allocated object, or, most frequently, by declaring objects to be components. This restriction is similar to proposals that allow updates on objects reachable from unique pointers, but there are interesting differences. Several other techniques also forbid reading or have to account in their analysis for reading, as done, for example, in Fugue [4] and in work by Boyland [2]. It would be interesting to further understand any relation between our allowing reading and the readonly references in universe types [15]. We expect that our technique can be extended to deal with richer heap structures as well.
- *Theory and practice.* We intend to implement the system, and may have a system running before the workshop starts. We also hope to do a formal proof of soundness.
- *Inference.* We believe that a practical system can only be realized if it supports some inference of specification fragments. Given class invariants, the existing approach seems well suited to infer some components, preconditions, and inductive invariants.

References

1. Amal Ahmed and David Walker. The logical approach to stack typing, March 2003.
2. John Boyland. Alias burying: Unique variables without destructive reads. 31(1):533–553, January 2001.
3. Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software, May 2001.

4. Robert DeLine and Manuel Fähndrich. The Fugue protocol checker: Is your software baroque? Submitted manuscript, 2003.
5. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, December 1998.
6. Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.
7. Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming, May 2002.
8. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java, May 2002.
9. Bart Jacobs. Objects and classes, co-algebraically. In B. Freitag, C. B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer Academic Publications, 1996.
10. K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, September 2002.
11. K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects, May 2002.
12. K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. In Bart Jacobs, Gary T. Leavens, Peter Müller, and Arnd Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*, Technical Report 251. Fernuniversität Hagen, May 1999. Also available as Technical Note 1999-002, Compaq Systems Research Center.
13. Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262. Springer-Verlag, 2002. PhD thesis, FernUniversität Hagen.
14. Peter Müller and Arnd Poetzsch-Heffter. Modular specification and verification techniques for object-oriented software components. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 7, pages 137–159. Cambridge University Press, 2000.
15. Peter Müller and Arnd Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, FernUniversität Hagen, 2001.
16. Greg Nelson. A generalization of Dijkstra’s calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, October 1989.
17. David Walker, Karl Crary, and Greg Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, July 2000.

An Assertional Proof System for Multithreaded Java^{*}

June 29, 2003

Erika Ábrahám¹, Frank S. de Boer²,
Willem-Paul de Roever¹, and Martin Steffen¹

¹ Christian-Albrechts-University Kiel, Germany

² CWI Amsterdam, The Netherlands

Abstract. In this paper we report on a tool-supported *assertional proof method* for a concurrent sublanguage of *Java*, covering shared-variable concurrency via instance variables, coordination via reentrant synchronization monitors, synchronous message passing, and dynamic thread creation. The verification method is formulated in terms of proof-outlines, where the assertions are layered into local ones specifying the behavior of a single instance, and global ones taking care of the connections between objects.

1 Introduction

The semantical foundations of *Java* [7] have been thoroughly studied ever since the language gained widespread popularity (see e.g. [3, 14, 5]). Besides standard object-oriented features, *Java* integrates multithreading and monitor synchronization. The research concerning *Java*'s proof theory mainly concentrated on various aspects of *sequential* sub-languages (see e.g. [10, 15, 13]). In this paper we present a tool-supported assertional proof system for *Java*'s monitor concept. The proof system generates verification conditions for *Java_{MT}*, a subset of *Java*, featuring dynamic object creation, method invocation, object references with aliasing, and, specifically, concurrency and *Java*'s monitor discipline.

The behavior of a *Java_{MT}* program results from the concurrent execution of methods. To support a clean interface between internal and external object behavior, *Java_{MT}* does not allow qualified references to instance variables. As a consequence, shared-variable concurrency is caused by simultaneous execution within a single object, only, but not across object boundaries. In order to capture program behavior in a modular way, the assertional logic and the proof system are formulated at two levels, a local and a global one. The local assertion language describes the internal object behavior. The global behavior, including the communication topology of the objects, is expressed in the global language. As in the Object Constraint Language (OCL) [16], properties of object-structures are described in terms of a navigation or dereferencing operator.

^{*} Part of this work has been financially supported by IST project Omega (IST-2001-33522) and NWO/DFG project Mobi-J (RO 1122/9-1, RO 1122/9-2).

The assertional proof system for verifying safety properties of *Java_{MT}* is formulated in terms of *proof outlines* [12], i.e., of programs augmented by auxiliary variables and annotated with Hoare-style assertions [6, 8]. The satisfaction of the program properties specified by the assertions is guaranteed by the verification conditions of the proof system. The execution of a single method body in isolation is captured by standard *local correctness* conditions, using the local assertion language. Interference between concurrent method executions is covered by the *interference freedom test* [12, 11], formulated also in the local language. It has especially to accommodate for reentrant code and the specific synchronization mechanism. Possibly affecting more than one instance, communication and object creation is treated in the *cooperation test*, using the global language. The communication can take place within a single object or between different objects. As these cases cannot be distinguished syntactically, our cooperation test combines elements from similar rules in [4] and in [11] for CSP.

Computer-support is given by the tool *Verger* (*VERification condition GENerator*), taking a proof outline as input and generating the verification conditions as output. We use the interactive theorem prover PVS to verify the conditions.

In the following we informally describe the language and the proof system. For formal syntax, semantics, verification conditions, soundness, and completeness we refer to [1].

2 The programming language *Java_{MT}*

Java_{MT} is a strongly typed sublanguage of *Java*. The abstract syntax is summarized in Table 1. Programs are collections of classes containing method declarations. *Instances* of the classes, i.e., *objects*, are dynamically created, and communicate via *method invocation*. As we focus on the concurrency aspects of *Java*, all classes in *Java_{MT}* are thread classes in the sense of *Java*: They contain a *start*-method which can be invoked only once for each object. The method spawns a new thread of execution which begins its live in the *run*-method while the initiating thread continues its own execution.

As a mechanism of concurrency control, methods can be declared as *synchronized*. Each object has a *lock* which can be owned by at most one thread. Synchronized methods of an object can be invoked only by a thread that owns the lock of that object. Without the lock, a thread has to wait until the lock becomes free. The owner of an object's lock can recursively invoke several synchronized methods of that object, which corresponds to the notion of reentrant monitors. Besides synchronization, objects offer the methods *wait*, *notify*, and *notifyAll* as means to facilitate efficient thread coordination at the object boundary. All four methods are parameterless and without a return value. A thread owning the lock of an object can block itself and free the lock by invoking *wait* on the given object. The blocked thread can be reactivated by another thread via the object's *notify*-method; the reactivated thread must re-apply for the lock before it may continue its execution. The method *notifyAll*, finally, generalizes *notify* in that it informs all threads blocked on the object.

$$\begin{aligned}
exp &::= x \mid u \mid \text{this} \mid \text{null} \mid f(exp, \dots, exp) \\
exp_{ret} &::= \epsilon \mid exp \\
stm &::= x := exp \mid u := exp \mid u := \text{new}^c \\
&\quad \mid u := exp.m(exp, \dots, exp) \mid exp.m(exp, \dots, exp) \\
&\quad \mid \epsilon \mid stm; stm \mid \text{if } exp \text{ then } stm \text{ else } stm \text{ fi} \mid \text{while } exp \text{ do } stm \text{ od } \dots \\
modif &::= \text{nsync} \mid \text{sync} \\
meth &::= modif m(u, \dots, u) \{ stm; \text{return } exp_{ret} \} \\
meth_{run} &::= \text{nsync run}() \{ stm; \text{return} \} \\
meth_{wait} &::= \text{nsync wait}() \{ ?\text{signal}; \text{return}_{getlock} \} \\
meth_{notify} &::= \text{nsync notify}() \{ !\text{signal}; \text{return} \} \\
meth_{notifyAll} &::= \text{nsync notifyAll}() \{ !\text{signal_all}; \text{return} \} \\
meth_{predef} &::= meth_{start} \ meth_{wait} \ meth_{notify} \ meth_{notifyAll} \\
class &::= c \{ meth_{start} \dots meth_{wait} \ meth_{run} \ meth_{predef} \} \\
class_{main} &::= class \\
prog &::= \langle class \dots class \ class_{main} \rangle
\end{aligned}$$
Table 1. $Java_{MT}$ syntax
$$\begin{aligned}
e &::= z \mid x \mid u \mid \text{this} \mid \text{null} \mid f(e, \dots, e) & e \in LExp \\
p &::= e \mid \neg p \mid p \wedge p \\
&\quad \mid \exists z. p \mid \exists z \in e. p \mid \exists z \sqsubseteq e. p & p \in LAss \\
E &::= z \mid \text{null} \mid f(E, \dots, E) \mid E.x & E \in GExp \\
P &::= E \mid \neg P \mid P \wedge P \mid \exists z. P & P \in GAss
\end{aligned}$$

Table 2. Syntax of assertions

3 The assertion language

The assertion logic consists of a *local* and a *global* sublanguage (cf. Table 2). The *local* assertion language is used to annotate methods in terms of their local variables and of the instance variables of the class to which they belong. The *global* assertion language describes a whole system of objects and their communication structure. In the assertion language we add the type **Object** as the supertype of all classes, and we introduce *logical variables* z different from all program variables. Logical variables are used for quantification and as free variables to represent local variables in the global assertion language.

Assertions are built using the usual constructs from predicate logic, where only the difference between the local and the global level deserves mention which concerns the form of quantification. In the local language, unrestricted quantification $\exists z.p$ is solely allowed for integer and boolean domains, but not for reference types, as for those types the range of quantification dynamically depends

on the *global* state, something which is not available on the local level. This does not mean one cannot speak about the existence of objects on the local level at all as long as one avoids reference to the global state. To be able to do so, the local language syntactically contains two restricted forms of quantifications where the range of quantification is explicitly mentioned in the formula. The existential formula $\exists z \in e.p$ asserts that p holds for *some object from e* , where the expression e evaluates locally to a list of objects. Similarly for $i\exists z \sqsubseteq e.p$, where $\sqsubseteq$ is meant to denote all subsequences of e . Global assertions are evaluated in the context of a global state. Thus, unrestricted quantification is allowed for all types and ranges over the set of *existing* values. Qualified references may be used in the global language only. We use $\models_{\mathcal{L}} p$ and $\models_{\mathcal{G}} P$ to state that the assertions p and P are satisfied in all contexts.

4 The proof system

The proof system has to accommodate for dynamic object creation, shared-variable concurrency, aliasing, method invocation, synchronization, and, especially, reentrant monitors and thread coordination. The following section shows how to augment and annotate programs resulting in proof outlines, before Section 4.2 describes the proof method.

4.1 Proof outlines

For a complete proof system it is necessary that the transition semantics of *Java_{MT}* can be encoded in the assertion language. As the assertion language reasons about the local and global states, we have to *augment* the program with fresh auxiliary variables to represent information about the control points and stack structures within the local and global states. An augmentation extends a program by assignments to auxiliary variables, which we call *observations*. The additional assignments are executed atomically as multiple assignments $\vec{y} := \vec{e}$. Observations are, in general, “attached” to statements they observe in a non-interleavable manner. This is syntactically represented using $\langle \vec{y} := \vec{e} \rangle^i$ which attaches the observation to the preceding statement. Though the reception of a method call is not represented by any statements, we use $\langle \vec{y} := \vec{e} \rangle^i$ at the beginning of method bodies also to denote the corresponding observation of the callee. As method calls $u := e_0.m(\vec{e})$ conceptually consist of two steps — handing over the parameters and reception of the result being stored in u — we need an additional form to observe atomically the reception of the return value. This form is represented as $\langle \vec{y} := \vec{e} \rangle^e$. A stand-alone observation not attached to any statement is written as $\langle \vec{y} := \vec{e} \rangle$; it can be inserted at any point in the program. Points between a statement and its observation are no *control points*, since they are executed in a single computation step; we call them *auxiliary points*.

The augmentation does not influence the control flow of the program but enforces a particular scheduling policy. An assignment statement and its observation are executed simultaneously. Object creation and its observation are executed

in a single computation step, in this order. For method call, communication, sender, and receiver observations are executed in a single computation step, in this order.

In the following we introduce some built-in auxiliary variables used to formulate the verification conditions. An important point of the proof system to achieve completeness is the identification of communicating objects and threads. We identify a thread by the object in which it has begun its execution. We use the type **Thread** thus as abbreviation for the type **Object**. This identification is unique, since an object's **start**-method can be invoked only once. During a method call, the callee thread receives its own identity as an auxiliary formal parameter **thread**.

A local configuration is identified by the object in which it executes together with the value of its auxiliary local variable **conf** storing a unique object-internal identifier. Its uniqueness is assured by the auxiliary instance variable **counter**, incremented for each new local configuration in that object. The callee receives the “return address” as auxiliary formal parameter **caller** of type $\mathbf{Object} \times \mathbf{Int} \times \mathbf{Thread}$, storing the identities of the caller object, the calling local configuration, and the caller thread. Note that the thread identities of caller and callee are the same in all cases but the invocation of a **start**-method.

To capture mutual exclusion and the monitor discipline, the instance variable **lock** of type $\mathbf{Thread} \times \mathbf{Int}$ stores the identity of the thread who owns the lock, if any, together with the number of synchronized calls in the call chain. Its initial value $(\text{null}, 0)$ indicates that the lock is free. The instance variables **wait** and **notified** of type $\text{list}(\mathbf{Thread} \times \mathbf{Int})$ store the threads waiting at the monitor, respectively those having been notified. Besides the thread identity, the number of synchronized calls is stored. In other words, these variables remember the old lock-value prior to suspension which is restored when the thread becomes active again. The boolean instance variable **started**, finally, remembers whether the object's **start**-method has already been invoked.

To specify invariant properties of the system, the augmented programs are *annotated* by attaching pre- and postconditions to all program statements. We use the triple notation $\{p\} \text{stm} \{q\}$ and write $\text{pre}(\text{stm})$ and $\text{post}(\text{stm})$ to refer to the pre- and the post-condition of a statement. For assertions at auxiliary points we use the following notation: The annotation

$$\{p_0\} u := e_0.m(\vec{e}); \quad \{p_1\}^i \langle \vec{y}_1 := \vec{e}_1; \rangle^i \quad \{p_2\}^{\text{ret}} \quad \{p_3\}^2 \langle \vec{y}_4 := \vec{e}_4; \rangle^2 \quad \{p_4\}$$

assigns p_0 and p_4 as pre- and postconditions to the method invocation; p_1 is assumed to hold directly after method call, but prior to its observation; p_2 describes the control point of the caller after method call and before return; finally, p_3 specifies the state directly after return but before its observation. The annotation of method bodies $\text{stm}; \text{return } e$ is as follows:

$$\{p_1\}^i \langle \vec{y}_2 := \vec{e}_2; \rangle^i \{p_2\} \quad \text{stm}; \quad \{p_3\} \quad \text{return } e; \quad \{p_4\}^i \langle \vec{y}_3 := \vec{e}_3; \rangle^i \{p_5\}$$

The callee postcondition of the method call is p_2 ; the callee pre- and postconditions of return are p_3 and p_5 . The assertions p_1 resp. p_4 specify the states of

the callee between method call resp. return and its observation. Annotation of object creation is similar.

Besides pre- and postconditions, for each class c , the annotation defines a local assertion I_c called *class invariant*, which may refer only to the instance variables of c , and which expresses invariant properties of instances of the class, i.e., properties required to hold for all instances of the class at each point during the execution.³ We require that for each method of a class, the class invariant is the precondition of the method body. Finally, a global assertion GI called the *global invariant* specifies properties of communication between objects. As such, it should be invariant under object-internal computation. For that reason, we require that for all qualified references $E.x$ in GI with E of type c , all assignments to x in class c occur in the observations of communication or object creation. We require that in the annotation no free logical variables occur. An augmented and annotated program is called a *proof outline* or *asserted program*.

4.2 Verification conditions

The proof system formalizes a number of *verification conditions* which inductively ensure that for each reachable configuration, the local assertions as well as the global and the class invariants hold. The conditions are grouped, as usual, into initial conditions, and for the inductive step into local correctness and tests for interference freedom and cooperation. Here we only informally describe the latter two.

4.2.1 The interference freedom test Invariance of local assertions under computation steps in which they are not involved is assured by the proof obligations of the *interference freedom test*. Since $Java_{MT}$ does not support qualified references to instance variables, we only have to deal with invariance under execution within the *same* object. Due to restrictions on the programming language, communication and object creation do not change the instance states of the executing objects. Implicitly, we only have to cover invariance of assertions annotating control points over assignments, including observations. Preconditions of observations do not have to be shown invariant, since the observations are executed together with the statements they observe in a single computation step, i.e., the preconditions of observations describe points at which no interleaving can take place. To distinguish local variables of the assertion from those of the assignment we rename the local variables in the assertion. We use primed assertions p' to denote a given assertion p with every local variable u replaced by a fresh one u' , and correspondingly for expressions.

³ The notion of class invariant commonly used for sequential object-oriented languages differs from our notion: In a sequential setting, it is sufficient that the class invariant holds initially and is preserved by whole method calls, but not necessarily in between. This weaker notion is adequate in a sequential language, whose semantics is based on the input/output behavior of programs, but not in a setting with shared variable concurrency, where also intermediate states are relevant for the behavior.

Generally, the interference freedom test defines for each assertion p at a control point and each assignment $\vec{y} := \vec{e}$ in the same class c a condition of the form $\models_{\mathcal{L}} \text{pre}(\vec{y} := \vec{e}) \wedge p' \wedge I_c \wedge \text{interleavable} \rightarrow p'[\vec{e}/\vec{y}]$, where the effect of assignments is expressed by the usual capture-avoiding substitution. The additional antecedent *interleavable* is explained in the following: An assertion p has to be invariant under an assignment $\vec{y} := \vec{e}$ only if the local configuration described by the assertion is not active in the computation step executing the assignment. If p and $\vec{y} := \vec{e}$ belong to the *same* thread, i.e., $\text{thread} = \text{thread}'$, then the only assertions endangered are those at control points waiting for a return value earlier in the current execution stack. Invariance of a local configuration under its own execution, however, need not be considered and is excluded by requiring $\text{conf} \neq \text{conf}'$. Interference with the *matching* return statement in a self-communication need also not be considered, because communicating partners execute simultaneously; in this case we additionally require $\text{this} \neq \text{caller_obj} \vee \text{conf}' \neq \text{caller_conf}$, where caller_obj is the first and caller_conf the second component of caller . If the assertion and the assignment belong to *different* threads, interference freedom must be shown in any case except for the self-invocation of the *start*-method.

Example 1. Assume an assignment $\{p\} \text{stm}$ in a method m of c , and an assertion q at a control point, which is not waiting for return, in the same method such that both p and q imply $\text{thread} = \text{this}$, i.e., the method is executed only by the thread of the object to which it belongs. Then, for invariance of q under the assignment stm , the antecedent of the interference freedom condition implies $p \wedge q' \wedge \text{interleavable}(q, \text{stm})$. From $p \wedge q'$ we conclude $\text{thread} = \text{thread}'$, and thus by the definition of $\text{interleavable}(q, \text{stm})$ the assertion q should be at a control point waiting for return, which is not the case, and thus the antecedent of the condition leads to a contradiction.

Example 2. This example shows how we can use the fact, that at most one thread can own the lock of an object, to show mutual exclusion. We use the assertion $\text{owns}(\text{thread}, \text{lock})$ for $\text{thread} \neq \text{null} \wedge \text{thread}(\text{lock}) = \text{thread}$, where $\text{thread}(\text{lock})$ is the first component of the lock value. Let furthermore $\text{free_for}(\text{thread}, \text{lock})$ be $\text{thread} \neq \text{null} \wedge (\text{owns}(\text{thread}, \text{lock}) \vee \text{lock} = \text{free})$.

Let q , given by $\text{owns}(\text{thread}, \text{lock})$, be an assertion in a class and let $\{p\}^i \langle \vec{y} := \vec{e} \rangle^i$ with $p \stackrel{\text{def}}{=} \text{free_for}(\text{thread}, \text{lock})$ be the callee observation at the beginning of a synchronized method in the same class. Note that built-in augmentation in $\vec{y} := \vec{e}$ changes the lock value setting the lock owner to the executing thread. Invariance of q under the observation $\vec{y} := \vec{e}$ is assured by the interference freedom condition $\models_{\mathcal{L}} p \wedge q' \wedge \text{interleavable}(q, \vec{y} := \vec{e}) \rightarrow q'[\vec{e}/\vec{y}]$. The assertions p and q' imply $\text{thread} = \text{thread}'$. After the execution of the built-in augmentation setting the lock owner to the identity of the executing thread in $\vec{y} := \vec{e}$ we have $\text{owns}(\text{thread}, \text{lock})$, i.e., $\text{owns}(\text{thread}', \text{lock})$, which was to show.

4.2.2 The cooperation test Whereas the interference freedom test assures invariance of assertions under steps in which they are not involved, the *cooperation test* deals with inductivity for communicating partners, assuring that

the global invariant, the preconditions, and the class invariants of the involved statements imply their postconditions after the joint step. Additionally, the preconditions of the corresponding observations must hold immediately after communication. The global invariant expresses global properties of the auxiliary instance variables which can be changed by observations of communication, only. Thus the global invariant is automatically preserved by the execution of non-communicating statements. For communication and object creation, however, the invariance must be shown as part of the cooperation test.

Here we describe only the cooperation test for method invocation. Since different objects may be involved, the cooperation test is formulated in the global assertion language. Local properties are expressed in the global language via substitution. To avoid name clashes between local variables of the partners, we rename those of the callee.

Let z and z' be logical variables representing the caller, respectively the callee object in a method call. We assume the global invariant, the class invariants of the communicating partners, and the preconditions of the communicating statements to hold prior to communication. For method invocation, the precondition of the callee is its class invariant. That the assertions indeed represent communicating partners and that the communication is enabled is captured in the additional antecedent **comm**. In case of a synchronized method invocation, the lock of the callee object has to be free or owned by the caller, which is expressed by $z'.\text{lock} = (\text{null}, 0) \vee \text{thread}(z'.\text{lock}) = \text{thread}$, where **thread** is the caller-thread and $\text{thread}(\alpha, n) = \alpha$. For the invocation of the monitor methods the executing thread must hold the lock. An additional predicate $e'_0[z/\text{this}] = z'$ in the condition of a method call $e_0.m(\vec{e})$ states, that z' is indeed the callee object where $e'_0[z/\text{this}]$ is the local assertion e'_0 expressed in the global language. The effect of communication and its observations are expressed by substitutions.

The verification conditions assure, that under the above assumptions, the preconditions of the corresponding caller and callee observations hold immediately after communication, and that the postconditions of the observations hold after their execution. The effects of communication and observation are expressed by substitutions.

For returning from a method we have similar conditions. Returning from the **wait**-method assumes that the thread has been notified and that the callee's lock is free. Due to the built-in auxiliary variables, method invocation hands over the "return address". Furthermore, due to some restrictions, the values of formal and actual parameters do not change during method execution. Therefore, they can be used in the return conditions to identify partners being in caller-callee relationship.

Example 3. This example shows how we can prove properties of parameter passing. Let $\{p\} e_0.m(v)$, with p given by $v > 0$, be a statement in a class c with e_0 of type c' , and let method $m(u)$ of c' have the body $\{q\} \text{stm}; \text{return}$ where q is $u > 0$. Inductivity of the proof outline requires that if p is valid prior to the call (besides the global and class invariants), then the method's precondition q is satisfied

after the invocation. Omitting irrelevant details, the cooperation test requires $\models_{\mathcal{G}} p[z/\text{this}] \rightarrow q'[z'/\text{this}][v/u']$, which expands to $\models_{\mathcal{G}} v > 0 \rightarrow (u' > 0)[v/u']$.

Example 4. The following example demonstrates how we can express dependencies between instance states in the global invariant, and use this information in the cooperation test.

Let $\{p\} e_0.m()$, with p given by $x > 0 \wedge e_0 = o$, be a statement in a class c with e_0 of type c' and with x an integer instance variable and o an instance variable of type c' , and let method m of c' have the body $\{q\} stm; \text{return}$ where q is $y > 0$ and y an integer instance variable. Let furthermore z be a logical variable of type c , and let the global invariant be given by $\forall z. (z \neq \text{null} \wedge z.o \neq \text{null} \wedge z.x > 0) \rightarrow z.o.y > 0$. Inductivity requires that if p and the global invariant are valid prior to the call, then the method's precondition q is satisfied after the invocation (again, we omit irrelevant details). The cooperation test requires $\models_{\mathcal{G}} GI \wedge p[z/\text{this}] \wedge \text{comm} \rightarrow q'[z'/\text{this}]$ which expands to

$$\begin{aligned} \models_{\mathcal{G}} & ((\forall z. (z \neq \text{null} \wedge z.o \neq \text{null} \wedge z.x > 0) \rightarrow z.o.y > 0) \wedge \\ & (z.x > 0 \wedge e_0[z/\text{this}] = z.o) \wedge (e_0[z/\text{this}] = z' \wedge z \neq \text{null} \wedge z' \neq \text{null})) \\ & \rightarrow z'.y > 0 \end{aligned}$$

Instantiating the quantification by z , the antecedent implies $z.o.y > 0 \wedge z' = z.o$, i.e., $z'.y > 0$.

Example 5. This example illustrates how the cooperation test handles observations of communication. Let $\{\neg b\} \text{this}.m() \{b\}^{ret}$ be a statement in a class c with auxiliary instance variable b and let m of c have the body $\{\neg b\}^t \{b := \text{true}\}^t \{b\} stm; \text{return}$. The cooperation test assures inductivity for the precondition of the observation. We have to show $\models_{\mathcal{G}} (\neg z.b \wedge \text{comm}) \rightarrow \neg z'.b$, i.e., as it is a self-call, $\models_{\mathcal{G}} (\neg z.b \wedge z = z') \rightarrow \neg z'.b$, which is trivially satisfied. The cooperation test requires for the postconditions $\models_{\mathcal{G}} \text{comm} \rightarrow (z.b \wedge z'.b)[\text{true}/z'.b]$ which expands to $\models_{\mathcal{G}} z = z' \rightarrow (z.b \wedge z'.b)[\text{true}/z'.b]$, whose validity is easy to see.

Example 6. Assume the invocation of a synchronized method m of a class c , where m of c has the body $\langle \vec{y} := \vec{e} \rangle^t \{\text{thread}(\text{lock}) = \text{thread}\} stm; \text{return}$. Note that the built-in augmentation in $\vec{y} := \vec{e}$ sets the lock owner. Omitting irrelevant details again, the cooperation test requires $\models_{\mathcal{G}} \text{true} \rightarrow (\text{thread}(z'.\text{lock}) = \text{thread}')[\vec{e}/\vec{y}]$, which holds due to the built-in augmentation, setting the lock owner.

5 Conclusion

The contribution reports on an assertional proof method for a multithreaded sublanguage of *Java* including its monitor discipline. It builds on earlier work, especially [2]). Currently we are implementing a translator, which, given an asserted *Java_{MT}* program, generates a number verification conditions to be handed-over to *PVS* for interactive verification. The underlying theory, the proof rules, and their soundness and completeness are presented in detail in [1].

As future work, we plan to extend $Java_{MT}$ by further constructs, like inheritance, subtyping, exceptions, etc. An extension of the semantics and the proof theory to detect deadlocks and termination is also of interest.

References

1. E. Ábrahám, F. de Boer, W.-P. de Roever, and M. Steffen. A Hoare logic for monitors in Java. Technical report TR-ST-03-1, Lehrstuhl für Software-Technologie, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Apr. 2003.
2. E. Ábrahám-Mumm, F. de Boer, W.-P. de Roever, and M. Steffen. Verification for Java's reentrant multithreading concept. In M. Nielsen and U. H. Engberg, editors, *Proceedings of FoSSaCS 2002*, volume 2303 of *Lecture Notes in Computer Science*, pages 4–20. Springer-Verlag, Apr. 2002. A longer version, including the proofs for soundness and completeness, appeared as Technical Report TR-ST-02-1, March 2002.
3. J. Alves-Foss, editor. *Formal Syntax and Semantics of Java*. LNCS State-of-the-Art-Survey. Springer-Verlag, 1999.
4. K. R. Apt, N. Francez, and W.-P. de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2:359–385, 1980.
5. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An event-based structural operational semantics of multi-threaded Java. In Alves-Foss [3], pages 157–200.
6. R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proc. Symp. in Applied Mathematics*, volume 19, pages 19–32, 1967.
7. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
8. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969. Also in [9].
9. C. A. R. Hoare and C. B. Jones, editors. *Essays in Computing Science*. International Series in Computer Science. Prentice Hall, 1989.
10. M. Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2001.
11. G. M. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15(3):281–302, 1981.
12. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.
13. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. Swierstra, editor, *Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 1999.
14. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine*. Springer-Verlag, 2001.
15. D. von Oheimb and T. Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In L.-H. Eriksson and P.-A. Lindsay, editors, *Proceedings of Formal Methods Europe: Formal Methods – Getting IT Right (FME'02)*, volume 2391 of *Lecture Notes in Computer Science*, pages 89–105. Springer-Verlag, 2002.
16. J. B. Warmer and A. G. Kleppe. *The Object Constraint Language: Precise Modeling With Uml*. Object Technology Series. Addison-Wesley, 1999.

Toward Automatic Generation of Provably Correct Java Card Applets

Alessandro Coglio

Kestrel Institute
3260 Hillview Avenue, Palo Alto, CA 94304, USA
Ph. +1-650-493-6871 Fax +1-650-424-1807
<http://www.kestrel.edu/~coglio>
coglio@kestrel.edu

Abstract. This paper overviews an ongoing project aimed at developing an automatic generator of Java Card applets from higher-level specification(s) written in a domain-specific language called “SmartSlang”. The generator is based on Specware, a system for the formal specification and refinement of software. The applet generator translates a SmartSlang spec into the logical language of Specware, re-expresses the translated spec in terms of Java Card concepts via a series of refinement steps using Specware’s machinery, and generates Java Card code from the refined spec. The Java Card concepts used for refinement and code generation are captured as a shallow embedding of the Java Card language and API in the logic of Specware. Since proofs are associated to refinement steps, the applet generator produces a machine-processable proof tree along with the code, enabling the correctness of the generated code (with respect to the spec) to be checked independently from the applet generator, via a smaller and simpler applet checker to be also developed in this project.

1 Introduction

A smart card [10] consists of a chip embedded in a plastic substrate that is the size of a credit card or smaller. Since the chip is very small, it has relatively limited storage and processing capabilities. A smart card communicates with the external world via metal contacts on the surface of the card or via an antenna wound into the card. The signals exchanged through the contacts or the antenna encode commands to and responses from the card. The card’s memory and functionality can be accessed exclusively via these commands and responses. Smart cards are easily portable, cost-effective computing devices that can securely store and process information. Applications include authentication, banking, and telephony. Since bugs in a smart card’s hardware or software may compromise its security, correctness is of paramount importance in smart cards.

Java Card [4] is a version of Java [2] for smart cards. A Java Card applet is written in a subset of Java and uses a different API from regular Java. The Java Card API provides functionality to receive commands and send responses according to certain standardized protocols, perform cryptographic computations,

etc. Java Card provides a uniform platform to develop portable smart card applications that can be also installed and updated after the card is issued to the user. In addition, the type safety of Java is a good foundation for security.

Even though Java is relatively high-level, developing Java Card applets requires the programmer to deal with fairly low-level details, thus greatly increasing the potential for bugs. For example, commands and responses must be explicitly decoded from and encoded into bytes. Furthermore, because of the severe memory limitations and the typical absence of garbage collection from Java Card implementations, often the programmer has to forgo object-oriented principles and write code that makes almost no use of inheritance, keeps nested method calls to a bare minimum, and allocates all the needed objects when the applet is installed, re-using them during the applet's normal operation. While smart card memory keeps increasing thanks to the continual advancements in semiconductor technology, the market demand for smart card functionality keeps increasing too. So, memory limitations are likely to be a problem at least for a while.

This paper overviews an ongoing project aimed at developing an automatic generator of space-efficient and time-efficient Java Card applets from higher-level spec(ification)s. The generator is being built on top of Specware [9], a system for the formal specification and refinement of software, whose mathematical foundations guarantee that the output applet code is provably correct with respect to the input applet spec. It is expected that the use of the generator will greatly increase the productivity of applet developers and the confidence that applets are correct.

2 Applet Generator

From the user's point of view, the applet generator is a box that takes an applet spec written in a domain-specific language called "SmartSlang" ("Smart Card Specification Language") and automatically produces Java Card code that implements the specified applet. Internally, there are three components that operate in sequence, as shown in Figure 1.

Fig. 1. Structure of the generator

2.1 SmartSlang and Translator

The design of (a first version of) SmartSlang is ongoing and not finalized yet. So, only some highlights and goals are discussed here.

The domain of the domain-specific language SmartSlang is the one of smart cards. This means that SmartSlang features high-level constructs for smart card concepts, such that an applet's functionality can be specified in a very clear and concise way, minimizing the chance of specification errors. SmartSlang includes explicit constructs for commands, responses, encoding of commands and responses as bytes forming Application Protocol Data Units (APDUs), error handling, cryptography (keys and encryption/decryption operations), personal identification numbers (PINs), challenge-response protocols, and so on. For example, commands are identified symbolically and are accompanied by declarations of their encoding as APDUs, as opposed to Java Card where APDU bytes are explicitly retrieved and decoded, dispatching to code that processes the various commands. As another example, cryptographic operations are readily accessible as pure functions operating on data and keys, as opposed to Java Card where a `Cipher` object is created, initialized with a key, and used to encrypt/decrypt data, possibly in chunks. Of course, the applet generator turns these higher-level constructs into Java Card code that implements their semantics.

SmartSlang features a richer type system than Java Card. For instance, instead of only the few Java Card integral types, arbitrary integer ranges can be used as types, to describe more precisely the semantics of certain values (e.g. a balance in an e-wallet cannot be negative). Conformance of values to integer range types, as well as all the other type safety properties of an applet spec (e.g. array indices within bounds), are meant to be statically checked by the applet generator. This requires a fancier type checker than found in typical languages. Since loops are generally problematic for this kind of analyses, the intention is to use restricted, tractable forms of loops (e.g. loop through an array) and/or to have the user add suitable annotations (e.g. invariants) to guide the type checker. The challenge is to develop a powerful but tractable type system that does not require general symbolic theorem proving (only lightweight decision procedures, e.g. for Presburger arithmetic) and does not force the user to write too many annotations. The payoff of full static checking is that more specification errors are caught (e.g. forgetting to check a balance in a debit command) and that the generated Java Card code will not throw any exception at run time.

SmartSlang has a mathematical semantics in terms of state machines: an applet is a state machine that updates its state and produces a response when it receives a command. Anyhow, SmartSlang should be simple enough to be effectively understood and used by smart card experts without a formal background.

We have used some tentative SmartSlang constructs to specify a few realistic applets, e.g. an applet that signs data with an internally stored private key after a successful PIN verification and/or challenge-response external authentication. In these preliminary experiments, the tentative SmartSlang specs are considerably shorter and clearer than their corresponding Java Card code.

The first component of the applet generator is a translator from SmartSlang to MetaSlang, the specification language of Specware. MetaSlang is a version of higher-order logic [1], similar to the languages of PVS [11], HOL [13], and other systems. A MetaSlang spec consists of some sorts (i.e. types), some op(eration)s (i.e. functions over the types), and some axioms that constrain the sorts and the ops, i.e. a spec is a theory in higher-order logic. The translation of a SmartSlang spec is a MetaSlang spec of the state machine denoted by the SmartSlang spec. In fact, this translation can be considered to formally define the semantics of SmartSlang. The translator also performs the needed type safety checks on the SmartSlang spec. The translator will be developed after the design of SmartSlang is finalized.

2.2 Refinement Engine

No substantial work has been done on this component yet. So, only high-level concepts are presented here.

The idea is that the high-level MetaSlang spec of the applet, obtained by translating the SmartSlang spec, is transformed into a low-level MetaSlang spec from which Java Card code is readily generated. The transformation consists of a series of refinement steps that are applied by means of Specware's underlying refinement machinery.

Specware provides a simple and powerful notion of refinement as a morphism between specs. A morphism is a mapping from the source spec's sorts and ops to the target spec's sorts and ops, such that all the axioms in the source spec are theorems in the target spec, once translated according to the sort/op mapping. Often a morphism maps the sorts and ops of the source spec to the same sorts and ops in the target spec, especially when the two specs describe the same entity at different levels of abstraction.

Starting from the high-level applet spec, each refinement step carried out by the refinement engine re-expresses the spec in lower-level terms, making use of Java Card concepts. For example, a state component consisting of a balance having some SmartSlang integer range type is re-expressed as having an appropriate Java Card type where the balance can fit without wasting space; e.g. for a balance between 0 and 100 a byte is sufficient, but for a balance between 0 and 1,000,000 two shorts (or just a short and a byte, if space is scarce) are needed (`int` can be used if supported by the target Java Card implementation). As another example, the application of a SmartSlang cryptographic operation is re-expressed as a series of actions where (a representation of) a `Cipher` object is created, initialized with some key, and used to encrypt/decrypt data, possibly in chunks.

2.3 Code Generator

A first version of this component has been completely designed and implemented. So, more details than for the other two components are given here.

At the end of the refinement process, the applet’s functionality is entirely expressed in terms of Java Card concepts. These Java Card concepts are captured in the form of parameterized (i.e. generic) MetaSlang specs that are suitably instantiated (i.e. particularized) for the specific applet. The code generator generates Java Card code from the instantiated specs.

The parameterized MetaSlang specs are a shallow embedding of (a significant subset of) the Java Card language and API in higher-order logic. The adjective “shallow” means that Java Card expressions and statements are identified with their semantics, i.e. their syntax is not explicitly formalized but is implicitly captured by MetaSlang terms. In contrast, in a “deep” embedding Java Card expressions and statements would be defined as syntactic entities to which semantics is associated. The choice of a shallow embedding is solely motivated by it being slightly simpler and smaller than a deep embedding; a deep embedding would work as well.

Figure 2 shows some excerpts of the formalized semantics of Java Card expressions and statements. Following the Java language specification [6], an expression may complete abruptly with an exception or normally returning a result that is a value, a variable (e.g. for assignment), or nothing (i.e. a call to a `void` method); a statement may complete normally or abruptly with an exception or returning a value (e.g. `return 2;`) or nothing (i.e. `return;`); other forms of statement abrupt completion are currently excluded from the formalization. In general, executing an expression or statement causes a side effect consisting of a state change and finite sequences of input/output events. An expression or statement is defined as the set of all its possible executions, where an execution consists of a side effect and a completion (note that a value of sort $S \rightarrow \text{Boolean}$ is isomorphic to a set of values of sort S). Expressions include calls to API methods, whose semantics is defined not in terms of Java Card constructs but directly, as if they were “macro-expressions” in the language.

Input and output events are a mildly original aspect of this formalization, compared to others. Certain API method calls cause APDU bytes to be transferred between the card and the external world. These APDU chunks are captured as input/output events. Having them explicit allows to express the input/output behavior of (the MetaSlang representation of) an applet’s code in terms of command and response APDUs.

Since the semantics is relational (i.e. sets of side effects and completions), expressions and statements may be non-deterministic. Non-determinism is used to model the creation of fresh references to new objects (the axioms only say that the fresh reference must not be already present in the heap) and to model API method calls for random number generation. Because of these two features, non-determinism ripples to all the expressions and statements of the formalization. Otherwise, expressions and statements could be defined as functions from old states and input events to new states, output events, and completions. An alternative approach is to capture fresh references and generated random numbers as input events, thus keeping the semantics functional.

```

sort ExpressionResult = | var Variable
                        | val Value
                        | nothing
sort ExpressionCompletion = | normal ExpressionResult
                           | abrupt Exception
sort StatementAbruptCompletion = | retnothing
                                | ret Value
                                | exc Exception
sort StatementCompletion = | normal
                           | abrupt StatementAbruptCompletion
sort SideEffect = {old : State,
                  new : State,
                  in  : FSeq InputEvent,
                  out : FSeq OutputEvent}
sort Expression = SideEffect * ExpressionCompletion -> Boolean
sort Statement = SideEffect * StatementCompletion -> Boolean

```

Fig. 2. Excerpts of Java Card expression and statement semantics

Java Card constructs are formalized by means of ops that combine subexpressions and substatements into compound expressions and statements. For example, addition is captured by an op of sort **Expression * Expression -> Expression**: the resulting expression combines the side effects of the subexpressions and yields the sum of their results as result, unless either subexpression throws an exception, in which case the compound expression completes abruptly with the same exception; for lack of space, we do not show the MetaSlang axioms here.

The parameters of the parameterized MetaSlang specs of Java Card consist of sorts and ops that are suitably instantiated to represent specific programs. For instance, one of the parameters is an uninterpreted sort **UserClass**, accompanied by an axiom stating its finiteness, whose values are meant to be exactly the user-defined classes comprising the program (API classes, which are fixed for every program, are captured by an interpreted sort **APIClass**). For instance, to represent a program with three user-defined classes **C**, **D**, and **E**, the sort is defined as consisting of three values:

```

sort UserClass = | class_C | class_D | class_E

```

Similarly, there are uninterpreted sorts for user-defined fields, methods, etc., as well as uninterpreted ops that associate fields and methods to the classes where they are declared, bodies to non-abstract methods, and so on. All these sorts and ops are accompanied by axioms that constrain their possible instantiations so that the instantiated specs represent legal programs (e.g. no method is both abstract and static).

In Specware, a parameterized spec is instantiated by means of a morphism from the subspec consisting of the parameter sorts and ops along with their constraining axioms, to a spec that instantiates those sorts and ops. The validity

of the morphism ensures that the parameters are instantiated consistently with their constraints, because all the axioms in the parameter subspec are theorems in the instantiating spec. The result of instantiation is the spec obtained by replacing the parameter subspec in the parameterized spec with the instantiating spec.

The refinement engine ultimately produces an instantiated spec that represents a specific Java Card program. The program is a Java Card applet that behaves as specified by the high-level MetaSlang spec obtained by translating the SmartSlang spec. The code generator examines the instantiated spec and straightforwardly produces Java Card source code for the program. For example, from the definition of the sort `UserClass`, the code generator determines which classes are declared in the program. Fields, methods, etc. are similarly determined from the definitions of the associated sorts and ops.

Code generation is reversible: from the Java Card program it is possible to reconstruct the instantiated spec from which the program was produced. For instance, the definition of the sort `UserClass` can be reconstructed by examining the classes declared in the program. The relevance of this reversibility is explained in the next section.

3 Independent Certification

Each refinement step carried out by the refinement engine of the applet generator has an associated proof that all the axioms in the source spec of the morphism are theorems in the target spec. The sequential composition of the morphisms corresponding to all the refinement steps yields a morphism from the high-level spec obtained by translating the SmartSlang spec to the low-level spec from which Java Card code is generated. The proof for the compound morphism is obtained by composing the proofs for the component morphisms.

A future goal of this project is to have the applet generator produce, besides the code, a machine-processable proof tree of the overall refinement, as shown in Figure 3. This enables to certify the correctness of the code with respect to the spec independently from the generator, by means of an automatic checker to be also developed in this project. The applet checker, also shown in Figure 3, takes as input the SmartSlang spec, the Java Card code, and the proof. The SmartSlang spec is translated into a high-level MetaSlang spec, using the same translator used in the applet generator. The Java Card code is translated into a low-level MetaSlang spec using the code abstractor, which is the inverse of the code generator used in the applet generator. Given the high-level and the low-level MetaSlang specs, a simple proof checker for the higher-order logic of MetaSlang is used to establish whether the purported proof is a valid proof that all the axioms in the high-level spec are theorems in the low-level spec.

When a Java Card applet is produced by the applet generator, the input spec and the generated proof constitute a checkable certificate that accompanies the generated code. By using the applet checker to establish the validity of the certificate, trust is essentially shifted from the relatively large and complex

Fig. 3. Independent certification

refinement engine to the much smaller and simpler proof checker. The size and complexity of the code abstractor are about the same as the code generator, relatively modest; the size and complexity of the translator are also relatively modest. The translator and code abstractor are necessary to bring the applet spec and the applet code, which are written in different languages (i.e. SmartSlang and Java Card), into a common logical language (i.e. MetaSlang) where their relationship is formally expressed and proved.

As mentioned above, independent certification is a future goal: no work has been done on generating refinement proofs or on the applet checker yet.

4 Implementation Language

The applet generator and checker are being written in MetaSlang. This is a natural choice because Specware is itself written in MetaSlang and because the applet generator and checker are built on top of Specware (e.g. they manipulate MetaSlang specs).

More in detail, the applet generator and checker are being written in an executable subset of MetaSlang, which is like a purely functional programming language. Specware is itself written in executable MetaSlang. Non-purely functional computation such as accessing files is encapsulated in small hand-written code fragments and/or in monads. Specware can turn executable MetaSlang into LISP, C, or Java. However, C and Java code generation are not as developed as LISP code generation yet. For this reason, the code generator component of the applet generator, as well as Specware itself, are currently generated in LISP.

Eventually, we plan to write high-level MetaSlang specs for the applet generator and checker and to obtain the executable ones via refinement. This will provide higher confidence in the correctness of the two tools.

5 Related Work

Hubbers, Oostdijk, and Poll [8, 7] have built a tool that automatically translates finite state machines into Java Card skeleton code and associated Java Modeling Language (JML) annotations [5]. The generated JML annotations express pre- and post-conditions, invariants, etc., which can be checked to hold for the generated code by means of existing tools, somewhat analogously to the certificate produced by our applet generator. A difference from our approach is that finite state machines, unlike SmartSlang, have no high-level constructs specific to smart cards. Another difference is that the generated skeleton code only implements (and accordingly the JML annotations only express properties of) part of the applet's functionality, essentially its control structure; the rest of the needed functionality must be added by hand. In contrast, our applet generator is meant to generate the complete code. However, those authors plan to extend their approach to generate more of the applet functionality.

Besides the work just mentioned, the author is unaware of other research projects aimed at automatically generating Java Card applets from higher-level specs.

There is extensive work on formally specifying the Java Card platform (language, virtual machine, and API) to precisely document and validate the platform itself and to verify properties of Java Card applets. That work relates to the formalization of the Java Card language and API used in the applet generator, but our formalization has a different purpose, namely to capture Java Card concepts for refinement and code generation.

Our formalization is relatively close to, and partially inspired by, the one developed within the LOOP project [3]. In that project, properties of Java Card applets are verified by translating the code into higher-order logic (like the code abstractor in the applet checker) and then using theorem provers like PVS and Isabelle [12] to prove conjectures in the resulting theories. The conjectures are expressed as JML annotations, which are translated into higher-order logic along with the code. The semantics of the Java Card language and API and of the JML annotations is formalized via a shallow embedding in higher-order logic in a coalgebraic style. The differences between the LOOP formalization and ours are not very profound; they are motivated by subjective reasons (e.g. relational vs. coalgebraic style) or by different project needs (e.g. our explicit input/output events to express the input/output behavior of applets). Our formalization covers a smaller Java Card subset than the LOOP formalization, but our coverage will grow as this project proceeds.

References

1. Peter Andrews. *Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Academic Press, 1986.
2. Ken Arnold, James Gosling, and David Holmes. *The Java™ Programming Language*. Addison-Wesley, third edition, 2000.

3. Bart Jacobs et al. The LOOP project. Information at <http://www.cs.kun.nl/~bart/LOOP>.
4. Zhiqun Chen. *Java CardTM Technology for Smart Cards*. Addison-Wesley, 2000.
5. Gary Leavens et al. The Java Modeling Language. Information at <http://www.cs.iastate.edu/~leavens/JML.html>.
6. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The JavaTM Language Specification*. Addison-Wesley, second edition, 2000.
7. Engelbert Hubbers and Martijn Oostdijk. Generating JML specifications from UML state diagrams. In *Proc. Forum on specification and Design Languages (FDL'03)*, September 2003.
8. Engelbert Hubbers, Martijn Oostdijk, and Erik Poll. From finite state machines to provably correct Java Card applets. In *Proc. IFIP SEC'03 Workshop on Small Systems Security (WG 11.2)*, May 2003.
9. Kestrel Institute and Kestrel Technology LLC. SpecwareTM. Information at <http://www.specware.org>.
10. W. Rankl and W. Effing. *Smart Card Handbook*. John Wiley and Sons, second edition, 2000.
11. SRI International. The PVS specification and verification system. Information at <http://pvs.csl.sri.com>.
12. Technical University of Munich and University of Cambridge. The Isabelle system. Information at <http://isabelle.in.tum.de>.
13. University of Cambridge. The HOL system. Information at <http://www.cl.cam.ac.uk/Research/HVG/HOL>.

Ownership: transfer, sharing, and encapsulation

Anindya Banerjee* and David A. Naumann **

Kansas State University and Stevens Institute of Technology

Abstract. Ownership confinement expresses encapsulation in heap structures, in support of modular reasoning about effects, representation independence, and other properties. Most previous proposals for static enforcement of ownership confinement require annotations of whole programs using intricate type systems that are difficult to validate. Moreover, few proposals deal with transfer and sharing of ownership. We show how consideration of a reasoning objective suggests a particular pattern of ownership amenable to lightweight and modular imposition of confinement. We show how the pattern extends to handle transfer, at the cost of a heavier analysis.

1 Introduction

For scalability in reasoning about programs, be it informal reasoning by developers or automated reasoning embodied in static analysis tools, modularity is essential. Modularity is achieved using encapsulation mechanisms such as visibility modifiers for object fields, but cannot be achieved fully in object-oriented programs without control of sharing of objects in the heap—for which we use the general term *confinement*. This paper reports on work towards a theory for reasoning about representation independence (i.e., class equivalence) for sequential Java programs on the basis of a flexible form of *ownership confinement*. To handle ownership transfer, it turns out that we also need an additional confinement property which is similar to the recent proposal called external uniqueness [9].

To be useful, a confinement discipline should (a) ensure some invariant that facilitates modular reasoning about some property of interest; (b) be sufficiently flexible to be applicable and useful for an interesting class of programs; and (c) be amenable to efficient static checking. Let us consider some example invariants, loosely described. *Uniqueness* is often used to strongly delimit effects, e.g., to avoid the need for a lock [16]: the invariant is that the object referenced from a unique field is not accessible from anywhere else. Boyapati et al. [5] use a form of *ownership* as basis for a locking discipline: if a thread holds a lock on a certain object, it need not lock objects owned by that object. Strong forms of ownership have been proposed for various kinds of modular reasoning [14, 15, 10, 7, 23, 20, 1]. A typical ownership invariant is that an owned object is not accessible except via the object that owns it. Ownership is ubiquitous, as it is

* e-mail:ab@cis.ksu.edu; Supported by NSF grants CCR-0209205 and CCR-0296182.

** e-mail:naumann@cs.stevens-tech.edu; Supported by NSF grant CCR-0208984.

a natural embodiment of aggregation and the encapsulation of representations. Achieving such encapsulation is a key design objective, as it supports local reasoning about mutable state, in the form of frame specifications (the “modifies” clause, frame rules) [19, 17], equivalence between versions of a class [3, 2], and general non-interference assertions [8, 4].

In previous work [3], we formulated a semantic notion of ownership and proved that it is a sufficiently strong invariant to justify *representation independence*, i.e., modular reasoning about equivalence of class implementations using the standard notion of simulation [18, 13, 21, 11]. This notion of ownership has several restrictions that make it inflexible in ways similar to some of the earlier proposals [14, 15]. One restriction is that encapsulated representation objects may not have *outgoing references* to clients of the owner. In the full version [2] of [3], this restriction was lifted and a static analysis given for confinement. The analysis is modular in the sense that constraints are imposed only on the owner class and its representation class(es).

The present paper extends our previous work by overcoming two other restrictions that have also been challenging for other work on ownership: multiple ownership and ownership transfer, i.e., transfer of representations between owners. A leading example of multiple ownership is collection classes, where a collection object encapsulates nodes of a data structure but allows access to those nodes by iterators. An example of ownership transfer is a group of collections, such as task queues among which tasks are transferred for load balancing.

2 Representation independence and ownership

We consider a simple example of representation independence, to show the significance of ownership confinement for showing equivalence between two versions of a class. This leads us to focus on a particular pattern of heap encapsulation which is applicable to a wide class of programs and which admits a lightweight static analysis (i.e., syntax-directed and requiring no program annotations).

Consider class `ListNode` in Fig. 1. Its instances are used in class `Fifo`; the nodes reachable from an instance of `Fifo` comprise the representation thereof. We use Java-like notation; in particular, class types are implicitly reference types. Here and throughout we consider fields to be private; methods are public unless otherwise indicated.

Privacy of fields `front` and `rear` helps encapsulate the objects that are intended to comprise an internal data structure which should not be directly accessible to clients of the queue. Not all reachable objects are within the representation, however; the contained items are not.

One might wish to reduce the overhead of object construction and destruction by substituting this data structure with another that uses an array. Provided that the public interface does not change, this concern should be local to class `Fifo`. To ensure that this change of representation does not affect the behavior of clients, the standard reasoning (based on simulation [13]) involves showing that clients do not depend on the internal representation, but depend only on the behavior

```

class ListNode {
  Object item; ListNode nxt;
  Object getIt(){ return item; }
  ListNode getNext(){ return nxt; }
  void setIt(Object o){ item := o; }
  void setNext(ListNode n){ nxt := n; } }
class Fifo { // owner of the nodes reachable from its fields
  ListNode front, rear; // least, most recent
  void enq(Object o){ ListNode n := new ListNode();
    n.setIt(o); if (front==null){ front := n; rear := n; } else { rear.setNext(n); rear := n; } }
  Object deq(){ ListNode n := front;
    if (front==rear){ front := null; rear := null; } else { front := front.getNext(); }
    return n.getIt(); }

```

Fig. 1. Toy example of FIFO queue.

of `Fifo` observed through its public methods. It is for this that confinement is needed: Without restrictions on aliasing to enforce encapsulation, *representation exposure* can occur. A client can depend on the representation, and indeed clients can interfere with the behavior of the queue.

Representation exposures violate various confinement properties one can think of. One realistic example similar to the toy queue example is the Java Class Signers bug. As discussed by Vitek and Bokowski [23], the type of the leaked object in this example has package scope. They propose that such objects should not escape from the package and they give a static analysis for package confinement; of course the example violates their static checking rules.

Package confinement has several limitations. Although package confinement is an invariant that delimits the scope of effects, and in that way supports modular reasoning, it is a rather large-scale or coarse-grained property. In particular, to use it for reasoning about substitution of one class implementation by another, one would have to reason about all instances of the class at once. But a programmer changing the representation for `Fifo` likely thinks in terms of a single instance. This is typical for objects that provide a collection or other data abstraction. Also, package confinement does not help with the situation where some public class, e.g., `HashTable`, is used for encapsulated representations but also for many other purposes. Our previous work [3] also failed to handle this situation.

For reasoning about class equivalence, as in the queue example, it is natural to focus on a single instance of the class to be replaced. Various forms of *ownership confinement* have been proposed, in which each instance of class `Fifo` can be viewed as *owning* its internal representation, which is not shared with other instances. To formulate a general rule for using simulation to prove equivalence between one implementation of a class and another, it is enough to focus on

- an instance of the class, say *Own*, to be replaced (e.g., a `Fifo` object)

Fig. 2. Confinement scheme for island j with respect to another island i . Dashed boxes delimit partition blocks. No objects are shown. Solid lines indicate allowed references and dotted lines indicate prohibited ones. There is no restriction within blocks.

- the internal representation for that instance —the encapsulated objects, called *reps*, on which clients should not depend, and which may be replaced. We assume these objects have a common superclass *Rep* (the generalization to a covering set of superclasses is straightforward).
- Objects that are neither owners nor representations thereof —which we lump under the term *clients*.

Thus we arrive at the view of the heap depicted in Fig. 2. This view has a simple characterization in terms of types. The set of objects in the heap is partitioned¹ into the following blocks:

- The client block, Ch , containing any object with type C such that $C \not\leq Own$ and $C \not\leq Rep$.
- Some *islands* of the form $Oh_i * Rh_i$ where Oh_i consists of a single owner object (with type C , $C \leq Own$) and Rh_i consists of the encapsulated representations for that owner (each of which has some type $D \leq Rep$).²

The invariant on the heap (called *heap confinement*) that must be maintained can now be formulated as restrictions on the direct points-to relation (see Fig. 2). A heap is confined provided there is a partition (as above) such that

- Clients do not point to encapsulated reps.
- Islands are separated from each other:
 - Owners do not point to reps in a different island.
 - Reps do not point to reps or owners in a different island.
- The pointers from Oh_i to Rh_i are in private fields of *Own*. Methods defined in a subclass may manipulate reps but not store them in its fields; this allows reasoning about versions of the owner class independently from its subclasses. (See [2] for examples of how this fits with the factory pattern.)

For instance-oriented reasoning about substitution of an owner class, our previous work [2] gives an *abstraction theorem* that formally justifies such reasoning

¹ We allow the “partition” blocks to be empty.

² We use the symbol $*$ for union of disjoint heaps, as in Separation Logic [22].

for programs that preserve confinement. A key point is that the basis for defining simulations—the hypothesis of the proof rule for class equivalence—is a relation connecting a single island for each of the two representations. Hence in light of Figure 2, we can sharpen the reasoning that justifies replacement of one class implementation by another: (a) The reasoner defines a *basic coupling* relation for a single island: that is, how an island for one implementation corresponds to an island for another. To connect, e.g., two versions of class `Fifo`, the basic coupling relates a single island $Oh * Rh$ for the first version (so Rh contains the instances of `ListNode` for a single queue) with an island $Oh' * Rh'$ for the second version (where Rh' might contain an array or whatever other data structure is used). (b) Next, for each method of the class, the reasoner proves the simulation property by showing that the corresponding implementations preserve not the coupling itself but the induced relation on complete heaps. This accounts for the fact that client objects and other owners can be reached from the island, while supporting per-instance reasoning. The simulation property for class `Fifo`, for example, is proved in terms of methods of `Fifo` executing on a single instance of `Fifo` (though behavior of a method may depend on objects elsewhere in the heap, due to outgoing calls to client methods).

The present paper pursues this approach further by exploring richer notions of confinement. To conclude this section we emphasize two points. First, confinement is in the eye of the reasoner. To reason about the connection between two implementations of a class, what matters is encapsulation of the entire representation, regardless of whether it might be possible to further decompose that representation into other ownership relations.³ (E.g., in a more object-oriented version, class `ListNode` could provide all the functionality of lists, via recursive calls to the tail, in which case one might view each node as owning the rest of the list. But for reasoning about `Fifo`, what matters is the behavior of operations on the list.)⁴

The second point is about static enforcement of confinement. Once we had found a semantic notion of confinement suitable for representation independence, we sought a static analysis, expecting that program annotations would be needed. But we were able to avoid that by making use of the program’s own types in formulating confinement and confinement rules. In the present paper, we improve our formulation and show the approach to be surprisingly flexible. But to some extent we contradict this second point: to handle transfer of ownership we use annotations for a form of uniqueness.

³ Thus we do not assume *hierarchical ownership* [7–9].

⁴ As an analogy, in the proof rule for loops what matters is that the body preserves the invariant; internal structure of the loop body is not relevant. Another analogy is the frame rule in Hoare logic, i.e., the inference of $\{P \wedge R\}S\{Q \wedge R\}$ from $\{P\}S\{Q\}$ (for suitable R). What matters is the top-level connective $\wedge$; of course R may have structure but this is not relevant to the rule. In Separation Logic, the rule is extended to the heap using $*$ as top-level connective.

Fig. 3. Refined confinement scheme; dotted arrows indicate disallowed references.

3 Sharing ownership

A well known challenge for ownership systems is posed by collections with which iterators may be associated. As an example, let us extend class `Fifo` with method

```
Enumeration enum(){ return new ListEnum(front); }
```

where `ListEnum` is an iterator.

An iterator needs access to the representation, so it is not possible to reason about it entirely separately. For the kind of example considered here, it seems natural to consider a single *primary owner*, the collection itself (of type $\leq Own$) together with a number of *sharing owners*, e.g., instances of `ListEnum` (we designate their supertype as *Osh*). This suggests the partition scheme depicted in Fig. 3, where each primary owner is in an island containing its reps and sharing owners. (Fig. 3 also includes a block Uh that is discussed later.) The island is still the unit of encapsulation used for purposes of reasoning about equivalence, where now the equivalence is between implementations of a pair of classes *Own*, *Osh* (e.g., `Fifo` and `ListEnum`). Heap confinement can again be formulated as restrictions on the direct points-to relation (see Fig. 3). The distinction between *Own* and *Osh* is important for formulating an instance-based notion of representation-independence, in the sense that one instance of an abstraction — defined by a class — is compared with another — given by an alternative version of the class. The extended abstraction involves multiple objects, the iterators, but only those for a single primary owner.

We are not the first to treat sharing of ownership; see [19, 5, 8]. A closely related work is that by Boyapati, Liskov, and Shriram [6] which gives an ownership system that allows sharing and which is claimed to be strong enough to support modular reasoning. As they remark, most other proposals are either too permissive for sound local reasoning or too restrictive to handle iterators. However, while [6] states that type soundness can be proved by standard techniques

```

void transferTo(Fifo o){ o.pushR(self.pullR()); }
void transferFrom(Fifo o){ self.pushR(o.pullR()); }
void pushR(o-unique ListNode n) /*module scope*/ {
  if (front==null){ front := n; rear := n; } else { rear.setNxt(n); rear := n; } }
o-unique ListNode pullR() /*module scope*/ {
  ListNode n := front;
  if (front==rear){ front := null; rear := null; } else { front := front.getNxt(); }
  n.setNext(null); result := n; }

```

Fig. 4. Transfer of rep between owners.

and proves that their ownership types ensure a certain heap invariant, there is no formalization of modular reasoning much less a justification that the ownership invariant helps. Also, the achievement of [6] comes at the cost of requiring inner classes for iterators; this is a reasonable program structure but complex and challenging for formal reasoning.⁵ Müller [19] handles a form of sharing and shows modular soundness for reasoning about modifies specifications; it is not clear how to adapt this result to representation independence.

All of the works cited above suffer from the shortcoming that the owner of an object is fixed. By contrast, our ownership invariant is a state predicate of the form: there is a partition of objects such that certain direct references do not exist. Our owners may traffic in reps.

4 Transfer of reps among owners

Owners may cooperate among themselves. For example, consider a number of queues of tasks, each serving its own processor. For purposes of load balancing, tasks from full queues can be transferred to empty ones. The public interface could be used to dequeue from one and enqueue to the other, but it could be difficult to do this and maintain task management information. Moreover, it incurs a performance penalty for object allocation and deallocation.

Figure 4 gives additional methods for class `Fifo`. Methods `transferTo` and `transferFrom` are for clients to use to transferring a rep from one `Fifo` instance to another. For example, the call `q2.transferTo(q1)` would be invoked to ask `q2` to transfer one of its tasks to `q1`.⁶ The other two methods are for use within the class, and are given package scope. Lea [16] identifies three forms of transfer via method calls: initiated by the recipient (e.g., `pullR`), initiated by the sender (e.g., `pushR`), and symmetric (exchange). Our example shows the first two, but exchange can be programmed similarly.

⁵ Another point of concern is that downcasts are omitted from [6] and it appears that treating them requires runtime support; downcasts and subclassing pose non-trivial challenges for ownership systems and representation independence, which are addressed in our work and in some others (e.g., [8, 1]) by purely static means.

⁶ Throughout the paper we omit both preconditions and error-checking code.

When is it safe to transfer a rep? As a first approximation we want *uniqueness*—the rep being transferred is referenced only by a unique pointer from the sending owner, say in island $Oh_j * Rh_j$, which is handed off to the receiving owner, in some island $Oh_i * Rh_i$. This ensures that putting the transferred rep in Rh_i does not create bad incoming pointers to Rh_i , such as might occur if the transferred rep had been referenced by an iterator in Oh_j . Outgoing pointers from the transferred rep to clients pose no problem. But if it had a pointer to the sending owner, or to an iterator in the sending island, the resulting state again violates confinement.

So we want a stronger property, *o-uniqueness*: a rep ℓ in $dom(Rh_j)$ may be transferred if the transferring owner in Ph_j has a unique pointer to ℓ . Moreover, consider the sub-island Rh'_j of Rh_j that is reachable from ℓ (note that ℓ is reachable from itself). We require that no locations of type $\leq Own$ or $\leq Osh$ be reachable from Rh'_j and that there be no pointers into Rh'_j from either Ph_j or Sh_j . In this case, the sub-island Rh'_j can be moved, *en masse*, to the destination island.

Although we have not completed the proofs, we are confident that these conditions suffice for a generalization of representation independence that is still based on a single-island simulation relation. The challenge is to prove, modularly, that after calling a method that may transfer reps, the caller’s environment (which may contain reps) satisfies the confinement conditions. In [2] we achieve this because the effect of any command, and thus any method meaning, on a confining partition is to *extend* it: objects may be added, but they do not move between islands.

The other major challenge is to impose o-uniqueness. Whereas straightforward syntactic rules, expressed only in terms of program types, suffice to ensure the confinement invariant in the absence of transfer, such rules do not seem feasible for o-uniqueness. The reason is that we must delimit reachable reps from among other reps. It seems unlikely to achieve a useful analysis that is “lightweight” in the sense of not using annotations.

As in our previous work, our approach is to start from the ultimate objective: a reasoning principle. Both confinement and o-uniqueness are semantic properties that suffice for a useful reasoning principle, namely a form of representation independence that does not require global reasoning about the heap. Although we have a satisfactory lightweight analysis for confinement, we factor out and leave unsolved the problem of static checking for o-uniqueness. Both our representation independence result and the soundness of our static analysis for ownership confinement are proved on the assumption of o-uniqueness at points designated by explicit annotations (method parameters and returns).

5 Transfer of reps between clients and owners

Leino et al [12] point out that in many cases, representation objects need to be initialized outside the owner class [12, 8]. Resource management (such as memory managers) poses the problem of transferring objects not only from clients to

encapsulating owners but also the reverse. Consider for example this manager of a toy resource `Rsrc`.

```
class Rsrc { Object it; Rsrc(Object i){ it:=i; } }
class RsrcMgr { // owner
  Rsrc freeList;
  RsrcMgr(int n){
    for (int i:=0; i<n; i++){ Rsrc r:= new Rsrc(freeList); freeList:=r; } }
  o-unique Rsrc alloc(){
    Rsrc r:= freeList; freeList:= (Rsrc)freeList.it; r.it:= null; return r; }
  void free(o-unique Rsrc r){ r.it:= freeList; freeList:= r; } }
```

To handle this, we refine our confinement scheme slightly, adding a single block of “unencapsulated reps”, Uh , as depicted in Fig. 3. (The term is misleading and should be replaced; they are client objects that happen to have type $\leq Rep$ and are subject to the additional restriction that they may not be pointed to by encapsulated reps.) This refinement deals with another shortcoming with our lightweight formulation based on program types. The previous schemes preclude the use of library types like `HashTable` which might be used both for the representation of *Own* and for other purposes by clients. We no longer insist that all objects of class *Rep* are used as reps for *Own* or *Osh*; rather, reps not in use that way are sufficiently separate to admit static checking.

In the example above, `alloc` allocates a confined rep object and “returns” it to the client. In the heap, a transfer occurs – this is why `alloc`’s return type is tagged **o-unique**. Following transfer, the object is no longer associated with the `freeList` but resides in Uh . As a result the owner no longer has access to it. Similarly, `free` disposes an unencapsulated rep object and returns it to the owner. In the heap, a transfer occurs – this is why `free`’s parameter is tagged **o-unique**. Following transfer, the object is no longer associated with Uh , but resides in some Rh_i . As a result a client no longer has access to it.

6 (Scant) Discussion

We do not believe there is a single ownership discipline —much less static analysis technique— useful for all purposes. But we do believe the ownership pattern studied here is widely applicable. Once we complete a rigorous proof that it ensures strong encapsulation, a rigorous proof that our static analysis is sound for the confinement property, and a fair and thorough survey of related work, we plan to extend the formal setting to encompass safe multithreading.

Confinement disciplines can and should be formally justified in terms of the reasoning benefit: the precise confinement invariant and the reasoning consequences (such as validity of a proof rule). In our experience it is easy to write down plausible rules and definitions and much harder to get them right. Many researchers including ourselves believe that confinement checking should be largely mechanized, so reasoners (be it the programmer or other tools) are freed to focus on other things; clearly such tools will be more useful if based on sound

rules. Our work demonstrates that it is neither intractable nor entirely tedious to prove soundness. This is not to deny the importance, even the primacy, of gaining empirical experience using prototypes, but rather to say that positive empirical results for a confinement proposal should not be the end of the story.

A number of different disciplines have been proposed in recent years; while some are simply improvements on predecessors, others are incomparable in both the confinement invariant and its intended application. This suggests there is no single “right” notion of ownership, but rather a number of useful patterns of program structure and reasoning. Confinement is in the eye of the reasoner. Arguably, this implies that rather than global program annotations in support of a particular confinement discipline and checker, lightweight disciplines and checkers are needed.

Acknowledgement: Thanks to the anonymous referees for their feedback on an earlier draft.

References

- Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *OOPSLA*, 2002.
- Anindya Banerjee and David A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. Revised and extended from [3]; submitted., 2002.
- Anindya Banerjee and David A. Naumann. Representation independence, confinement and access control. In *POPL*, pages 166–177, 2002.
- Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *CSFW*, pages 253–270, 2002.
- Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*, 2002.
- Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *POPL*, 2003.
- David Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, Australia, 2001.
- David Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, 2002.
- David Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *ECOOP*, 2003.
- David G. Clarke, James Noble, and John M. Potter. Simple ownership types for object containment. In *ECOOP*, 2001.
- Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.
- D. Detlefs, K. R. M. Leino, and G. Nelson. Wrestling with rep exposure. Technical Report 156, COMPAQ Systems Research Center, July 1998.
- C. A. R. Hoare. Proofs of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.
- J. Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA*, 1991.
- John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The Geneva Convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, 1992.
- Doug Lea. *Concurrent Programming in Java*. Addison-Wesley, second edition, 2000.
- K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5), 2002.
- R. Milner. An algebraic definition of simulation between programs. In *Proceedings of Second Intl. Joint Conf. on Artificial Intelligence*, pages 481–489, 1971.
- Peter Müller. *Modular Specification and Verification of Object-Oriented programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
- Peter Müller and Arnd Poetzsch-Heffter. A type system for controlling representation exposure in Java. In *ECOOP Workshop on Formal Techniques for Java Programs*. Technical Report 269, Fernuniversität Hagen, 2000.
- John C. Reynolds. Types, abstraction, and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing '83*, pages 513–523. North-Holland, 1984.
- John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS*, 2002.
- Jan Vitek and Boris Bokowski. Confined types in Java. *Software Practice and Experience*, 31(6):507–532, 2001.

Static Detection of Atomicity Violations in Object-Oriented Programs

Christoph von Praun and Thomas R. Gross

Laboratory for Software Technology
ETH Zürich
8092 Zürich, Switzerland

Abstract. Violations of atomicity are possible sources of errors in parallel programs. A violation occurs if the effect of a method execution depends on the execution of concurrent threads that operate on the same or overlapping parts of a shared data structure. All accesses to shared data are assumed to be ordered through synchronization, hence common techniques for data race and deadlock detection are not able to find such errors.

We have developed a static analysis that infers atomicity constraints and identifies potential violations. The analysis is based on an abstract model of threads and data. A symbolic execution tracks object locking and access and provides information that is finally used to determine potential violations of atomicity. We provide a detailed evaluation of our algorithm for a number of Java applications. Although the algorithm does not guarantee to find all violations of atomicity, our experience shows that our method is efficient and effective in determining several known synchronization problems in a set of application programs and the Java library. The problem of overreporting that is commonly encountered due to conservatism in static analyses is moderate.

1 Introduction

The use of locks and access to shared data in parallel programs entail the risk of errors that are not known in sequential programming: A possible source of error are violations of atomicity. A violation occurs if the effect of a method depends on the execution order of concurrent threads that operate on the same or overlapping parts of a shared data structure. Such a scenario is possible even if the data structure is protected through synchronization and individual accesses do not constitute a data race.

Atomicity is commonly understood as a property that is associated with statements and methods. Hence the search for violations of atomicity typically investigates the structure of statements and interleavings of threads. Flanagan and Qadeer [5, 4], e.g., have developed a type system that verifies the atomicity of methods. The type checker associates atomicities at the level of statements and combines these atomicities based on Lipton’s theory of left and right movers [6] to obtain atomicity information for statement groups and methods. This approach requires explicit information about the synchronization discipline and lock protection of shared variables that are typically provided by program annotations.

The goal of this work is to provide a fully automated whole program analysis that detects methods that violate atomicity and identifies the data structures and calling contexts that lead to the violation. We assume that programs are free from data races and that accesses to shared data are ordered through monitor-style synchronization.

Our approach is motivated by previous work of Artho et. al. [1] which analyzes the structure of locking in a program and infers consistency constraints (*view consistency*) from the program for sets of shared variables. Violations of this consistency model correspond to potential synchronization defects called *high-level data races*. The notion of high-level data races is similar to violations of atomicity although both concepts are incomparable and several important scenarios of atomicity violations are not covered by the definition of high-level data races [9].

A common observation in programs that exhibit high-level data races or atomicity violations is that shared data is accessed with incoherent synchronization: Consider the example of a bank account in Figure 1: method `update` is invoked by several concurrent `Update` threads. The shared variable `balance` is accessed under common lock protection and hence there is no data race. The structure of locking specifies that the lock associated with the `Account` instance protects *either* a read *or* a write of field `balance`. Method `update` applies this synchronization discipline, however it performs a read *and* a write and hence cannot be atomic.

```

class Account {
    int balance;
    synchronized int read() {
        return balance;
    }
    int update(int a) {
        int tmp = read();
        synchronized(this) {
            balance = tmp + a;
        }
    }
}

class Update extends Thread {
    void run() {
        Example.a.update(123);
    }
}

class Main {
    static Account a;
    static void main(String args[]) {
        a = new Account();
        new Update().start();
        new Update().start();
    }
}

```

Fig. 1. Example of an account class with non-atomic `update` method.

The example illustrates a common pattern of software defects where one thread first queries the state of some shared data structure and then relies on this information during the further execution while being oblivious to the fact that other concurrent threads could have changed the data structure in the meantime. More generally, such faults are subsumed under the term *atomicity violations*.

The detection of atomicity violations is a research issue and different type-based and dynamic methods have been proposed [5, 4, 1, 9]. We address this problem with a static whole program analysis that detects potential violations in Java programs.

The analysis is based on an abstract model of threads and data. A symbolic execution tracks object locking and access, and provides information about the synchronization discipline that is finally used to determine synchronization de-

fects. This algorithm is neither sound, i.e., there can be underreporting, nor complete, i.e., there can be overreporting. However, the algorithm detects all cases where one thread reads a shared variable under lock protection that may consequently be modified by concurrent threads (hence the result of the read might become stale). Our experience shows that this property covers a large number of cases of atomicity violations that have been reported earlier [5, 4, 9] and that overreporting is moderate (Section 4).

The focus of this paper is on the *static* detection of atomicity violations; other parts of the static analysis (that are, e.g., also used for escape analysis and static race detection) are mentioned only as far as necessary and are described in more detail in [7] and [8].

2 Method consistency

Method consistency specifies an access discipline for shared variables. The access discipline is determined from the access behavior of methods and the usage of locks. Method consistency adopts concepts from *view consistency* [1] and extends it to accommodate the scope of methods as consistency criterion. A violation of method consistency indicates a potential violation of atomicity at the method level. In some cases, such violations of atomicity are undesirable and represent software faults.

The rationale of method consistency is to conjecture atomic treatment for a set of shared variables that are accessed in the dynamic scope of a method (*method view*). The execution of a method is atomic if there are no concurrent updates of variables in its method view.

The activities of threads are modeled by *lock views*. A *lock view* is a set of *variable/access* pairs that correspond to variable accesses of a thread t in the dynamic scope of a lock. The *access* component specifies if the variable is read (**r**), or updated (**w**), i.e., written or read and written. There is one entry per variable. The set of lock views of a thread t is specified as $L_t = \{l_1, \dots, l_n\}$. For the program in Figure 1, lock views correspond to the fields accessed inside the synchronized method **read** and the synchronized block in method **update**: $L_{\text{update}} = \{\{\text{balance/r}\}, \{\text{balance/w}\}\}$.

The conjecture about sets of variables that should be treated atomically is given through method views m_i for each method i . The set of method views per thread t is $M_t = \{m_1, \dots, m_n\}$. There are two entries per variable, namely a read and a write entry. The method views for the **Update** threads in the example are $M_{\text{update}} = \{m_{\text{update}}\} = \{\{\text{balance/r}, \text{balance/w}\}\}$.

We need two concepts to define method consistency: (A) *view overlap* and (B) the *chain property*. (A) Two views v_i and v_j *overlap* if their intersection is not empty, i.e., $v_i \cap v_j \neq \emptyset$. (B) A set of views $\{v_1, \dots, v_n\}$ *forms a chain* with respect to a view v , if for all pairs of views v_i, v_j , of which at least one originates from a thread that is concurrent to the originating thread of v , holds $((v_i \cap v) \subseteq (v_j \cap v)) \vee ((v_j \cap v) \subseteq (v_i \cap v))$.

Method consistency exists, if, for all method views, the overlapping lock views form a chain. The concept of overlap serves to filter out irrelevant variables. The chain property detects lock usage scenarios that are susceptible to atomicity violations: e.g., a lock protects reads *or* updates of one variable or a lock protects

different but overlapping sets of variables (see high-level data races [1]). If method consistency is violated, a potential violation of atomicity is detected.

In the example, there are multiple concurrent threads t_{update} with a method view $m_{\text{update}} = \{\{\text{balance}/r, \text{balance}/w\}\}$. All lock views in L_{update} overlap with m_{update} , but they do not form a chain, hence method consistency is violated for method `update`.

3 Static analysis

The static detection of atomicity violations is based on a whole program analysis and is done in three steps. First, an abstract model of threads and heap data is computed (Section 3.1). Then a symbolic execution of abstract threads infers information about locking and object accesses (Section 3.2). Finally, claims of atomicity are established and validated (or refuted) (Section 3.3).

3.1 Modeling threads and data

In multi-threaded Java programs threads correspond to the execution of the `main` method and the `run` methods of objects that implement the interface `Runnable`. A compiler can determine *abstract threads* based on the allocation sites of `Thread` objects; the call graph of such threads is rooted at the `run` method of the thread object or an associated `Runnable` object. In many cases, a compiler cannot determine the actual number of runtime instances that originate at a thread allocation site; in this case, conservative assumptions are made and multiple concurrent runtime threads are assumed.

Java employs a simple memory model: Objects are allocated on a global heap and object access is possible only through references issued at object creation time. This model facilitates the approximation of the runtime object structure in a heap shape graph (HSG) [7] at compile time. Nodes in the HSG correspond to *abstract objects* and represent individual runtime objects or sets of objects that are aliased. Edges represent points-to relations introduced through reference fields. The overall result of the shape analysis is a set of graphs rooted at class or thread nodes.

For the purpose of determining view consistency, the construction of the HSG determines if abstract objects are potentially accessed from multiple runtime threads. Accesses to fields of such *shared* abstract objects during the symbolic execution (Section 3.2) constitute entries in views.

3.2 Symbolic execution

This section gives a brief overview on the symbolic execution that computes approximations of method views and lock views (Section 2) at compile-time. Details and optimizations are discussed in [8].

The symbolic execution analyzes individual instructions along the control-flow and call structure of an abstract thread; for multi-threaded programs, the analysis treats each abstract thread in a sequence. Before the analysis branches into a method invocation, the object context that the called method operates in is determined: For each local reference variable of the callee, the abstract object to which it refers is determined. For objects that are accessible to multiple threads, the abstract objects correspond to nodes in the HSG. Hence methods are

analyzed in object contexts, and object accesses during the symbolic execution can be attributed to the abstract objects they target. A single traversal of loops and recursion is sufficient to track accesses to abstract objects (each iteration would access the same set of abstract objects).

The structure of locking in Java allows the compiler to track abstract objects that are locked along the execution path in a stack. At object access sites, e.g., *getfield*, *putfield* bytecodes, information about the locked abstract objects, the accessed abstract object, and the accessed field are available. Hence views can be recorded similarly to the runtime procedure in [1]. In contrast to (runtime) views, the static analysis computes *abstract views* with respect to locked abstract objects; in particular, only accesses to objects that are potentially shared are recorded. In our current implementation, abstract views contain field symbols and we do not distinguish accesses to fields on different abstract objects.

3.3 Detecting atomicity violations

The procedure to detect violations of method consistency is based on abstract views and checks the criteria described in Section 2 (overlap, chain); accesses to final, volatile, and read-only fields are omitted from the views.

Method consistency is designed as an extension of view consistency. For languages like Java, where synchronization is often used at method boundaries, violations of view consistency [1] imply violations of method consistency. Wang and Stoller [9] note that view consistency and the absence of atomicity violations are incomparable. The same holds true for method consistency. Hence, according to the classification of *soundness* and *completeness* [3], our procedure to determine atomicity violations is *unsound* and *incomplete*.

An example for the unsoundness of our algorithm, i.e., an atomicity violation that is not detected, is given in Figure 2.

```

class Counter {
    int i;
    synchronized int inc(int a) {
        i += a;
        return i;
    }
}

class Main extends Thread {
    static Counter c;
    static void main(String args[]) {
        c = new Counter();
        new Main().start();
        new Main().start();
    }
    void run() {
        int i = c.inc(0);
        c.inc(i);
    }
}

```

Fig. 2. Example of underreporting: Although the lock views that overlap with the view of method `run` form a chain, the sequence of updates in the `run` method is not atomic and does not necessarily double the counter value.

Figure 3 illustrates an examples for the incompleteness of our algorithm, i.e., a report of a method inconsistency that does not correspond to a violation of atomicity.¹

So far, the conceptual capabilities of method consistency have been discussed. Additional imprecision is added through the fact that static analysis relies in

¹ The initialization pattern used in the example may not work as expected under current Java memory semantics; the memory model is being revised [2].


```

class Map {
    Object[] keys, values;
    Object[] values;
    boolean volatile init_done = false;
    void init() {
        if (!init_done)
            synchronized (this) {
                init_done = true;
                // update keys and values
            }
        synchronized Object get(...) {
            // read keys and values
            return ...
        }
    }
}

class MapClient extends Thread {
    static Map m;
    static void main(String args[]) {
        m = new Map();
        new MapClient().start();
        new MapClient().start();
    }
    void run() {
        // lazy initialization
        m.init();
        o = m.get(...);
        ...
    }
}

```

Fig. 3. Example of overreporting: The lock views of method `run` do not form a chain (lock protects reads *or* updates), hence method consistency is violated. However, the initialization of the map happens only once and the effect of method `run` is the same regardless of the thread interleaving.

many cases on a conservative approximation of the runtime situation. Abstract views might subsume a larger number of variables than actual runtime views due to infeasible control-flows or the inability of the static analysis to differentiate accesses to field variables in different object instances. The approximation of the static analysis to distinguish different object instances and to determine thread interference and data sharing can lead to reports that do not correspond to real violations of atomicity – hence a potential source of overreporting.

4 Experience

We have implemented the static analysis in a way-ahead Java compiler and use the GNU Java library version 2.96.

First, we verify if our analysis is capable to detect known violations of atomicity that correspond to synchronization defects. Our system determines atomicity violations that correspond to the scenarios of the `Account`, `StringBuffer`, `Vector` and `PrintWriter` classes in [4]. Moreover, non atomicity in the use of iterators for common collection classes like `Vector` and `Hashtable` that are discussed in [9] are detected. These classes provide explicit means to determine actual violations of atomicity at runtime (`ConcurrentModificationException`). We have also successfully checked several scenarios with high-level data races, e.g., the `Coordinate` example in [1].

Second, we look at several benchmark programs and determine potential violations of atomicity at the application scope: `philos` is a simple dining philosopher application, `elevator` a real-time discrete event simulation; `mtrt` is a multi-threaded raytracer, `tsp` a traveling salesman application, `hedc` a web meta-crawler, `specjbb` an e-commerce benchmark, and `jigsaw` a public domain web-server (version 1.0). All other benchmarks stem from the multi-threaded Java Grande Benchmark suite.

I/O facilities are often shared among threads and interaction sequences of individual threads are usually not atomic. We omit this common case and do not report violations of method consistency related to I/O library classes. Moreover, our current implementation does not account for accesses to arrays and hence

atomicity violations that are due to thread interference on shared arrays may not be reported.

Our implementation partitions views according to the affiliation of fields with classes. This means that the overlap and chain properties are determined only among field variables that belong to the same class. This strategy is justified because OO design typically imposes consistency constraints on variables of the same class; moreover, violations of the chain property for unrelated variables are omitted. We have not encountered a real synchronization defect that is overlooked due to this partitioning of views.

Accesses that occur during the initialization of a shared object or data structure cannot participate in inter-thread interference that leads to violations of atomicity. Hence we chose that views should not account for accesses through `this` in the scope of a constructor and for accesses in the scope of initializer methods. This convention is practical to reduce the number of spurious reports but entails the potential of underreporting.

The first columns in Table 1 show the execution times of the symbolic execution (*symexe*) and method consistency checking (*cons*) and the *memory* requirements of the static analysis on a P4 (1.4GHz) PC. Overall, the analysis is practical for the reported programs. The duration of the symbolic execution depends on the precision of type and alias information to narrow polymorphism. Programs like *jigsaw* and *specjbb* use dynamic class loading and instancing, which is modeled conservatively in the compiler and hence leads to imprecision. *hedc*, where conservative assumptions must be made due to a large recursion in the callgraph, is also negatively affected by conservative assumptions.

program	size	time[s]	time[s]	mem	reports		methods	
	LOC	symexe	cons	[MB]	app	lib	app	lib
philo	81	0.3	0.1	1	0	0	0/4	0/26
elevator	528	0.4	0.2	3	0/0/1/0	0/1/0/0	4/16	2/43
mtrt	11298	1.5	1.1	5	0	0/3/0/0	0/43	3/227
tsp	706	0.4	0.2	1	0/1/0/0	0	1/14	0/39
hedc	27952	140.8	19.9	58	0/1/4/0	3/7/6/0	11/135	3/350
specjbb	31903	62.1	23.9	30	0/17/0/0	0/3/3/0	19/447	5/248
jigsaw	31596	357.0	18.4	34	0/19/2/2	2/4/1/0	17/471	3/276
mol	1402	0.4	0.3	2	0/1/0/0	0/2/0/0	6/28	0/24
ray	1972	0.5	0.3	3	0/1/0/0	0/2/0/0	3/45	0/26
monte	3674	0.6	0.3	3	0	0/1/0/0	2/71	0/68
crypt	1241	0.1	0.1	2	0	0	0/10	0/1
lufact	1627	0.1	0.1	2	0	0	0/15	0/1
series	967	0.1	0.1	2	0	0	0/10	0/1
sor	876	0.1	0.1	3	0	0	0/7	0/1
sparse	868	0.1	0.1	2	0	0	0/8	0/1

Table 1. Analysis characteristics and reports of atomicity violations.

Further columns in Table 1 characterize violations of method consistency that we found. Column *reports* specifies the number of method views that are found to be inconsistent with lock views. We report only the smallest method views that still exhibit violations; method views that are supersets of those reported would exhibit the same violations but would make it more difficult to identify the cause of the report. If interference is due to field variables that belong to the library classes, numbers are reported in category *lib*, otherwise in category

app. A report contains information about (1) the methods that exhibit this view, (2) the class to which the fields in this view belong to, (3) the individual field variables in this view, (4) potentially interfering lock views (locked object or method and the set of variables that cause interference), and (5) the allocation sites of the object on which the interference occurs. For each entry in column *reports*, we partition the reports into *false/spurious/benign/harmful*:

False reports are due to the imprecision of the static analysis (e.g., if data is not shared but actually thread-local).

Spurious reports specify that violations of atomicity do not occur at runtime in the given usage context of a data structure due to higher level synchronization (e.g., through a protected encapsulating object or thread start/join; see also example in Figure 3).

Benign reports refer to situations where an atomicity violation at the method level is possible. Such situations are not uncommon and do not necessarily represent a synchronization fault. This is especially true for methods that are invoked at a high level in the caller hierarchy of a multi-threaded application with shared data. In many cases, violations of atomicity are a natural consequence of the use of shared variables for explicit inter-thread communication.

Harmful reports mean that a violation of atomicity may occur that may lead to unintended runtime behavior.

The individual assessment of reports can be difficult and requires precise information about the synchronization disciplines for the affected shared data structures, hence we use this classification schema as a guidance.

Column *methods* in Table 1 specifies the number of methods reported by the checker and the overall number of methods for which a view is registered. The reports contain only methods that (1) access variables in at least one subordinate lock view (view v_1 is subordinate to v_2 if v_1 occurs in the scope of v_2), and (2) that are at the lowest levels of the caller hierarchy. Aspect (1) suppresses reports of method that do not use synchronization during their execution but exhibit a method view that is conflicting with lock views. For those methods, we report their callers (one of those will make use of synchronization because we assume that there are no data races). Aspect (2) excludes the reporting of all callers of a method for which we determined a potential violation of atomicity (a method that calls a non-atomic method is not atomic either). If a method belongs to a library class, it is reported in category *lib*, else *app*.

Most of the smaller benchmarks share data in arrays, hence there are few or no classes that we consider for reporting. In *elevator*, there is one benign report for a shared data structure that represents the state of the simulated system and is repeatedly accessed by the top-level methods of the simulator threads. A spurious report concerns an instance of class **Vector** that is however used such that no concurrent modification can occur.

mtrt exhibits three spurious reports that concern **Vector** and **Hashtable** data structures used in the library; these data structures are initialized once and then read (the scenario is similar to Figure 3). *tsp* has one spurious report due to a lock scope that violates the chain property but actually executes without concurrency during the initialization of the program.

In `hedc`, four reports are false and correspond to execution scenarios that the compiler conservatively assumed due to imprecise type information. Similar to `mtrt`, several reports are spurious on shared collection classes where initialization and subsequent shared read are ordered. Some reports are benign, e.g., for variables that are used to communicate information between worker and controller thread; another benign report addresses methods that perform subsequent accesses to a shared thread pool.

In `specjbb`, 12 reports correspond to classes that represent database records, where fields are accessed independently and atomicity is only necessary at the level of individual fields, or explicitly ensured by the transaction logic that is implemented at the application level. Depending on the correctness criteria at the application level, these reports can be classified as spurious or benign. Three further reports are benign and concern shared data containers that hold database records.

We discuss two interesting reports for `jigsaw`. The first report addresses class `ClientState` that represents an element of a linked list of client connections. Its fields `prev` and `next` link the structure and are accessed independently from fields `idle` and `client` (lock views are disjoint). All fields are cleared when a connection is removed from the pool and hence the fields are combined to a method view, leading to a report that does not reflect a problem in the program. The second report concerns class `ResourceStoreManager` in Figure 4. Method `shutdown` intends to remove all entries from the store (map referenced through field `entries`) and prevent further insertions by setting the latch `closed`. Atomicity is violated for method `loadResourceStore` (the sequence `checkClosed` and `lookupEntry` is not atomic). An unfortunate schedule can lead to the situation that entries are added to a resource manager after method `shutdown` has executed.

```

class ResourceStoreManager {
    boolean closed = false;
    Map entries = new HashMap();
    synchronized void checkClosed() {
        if (closed)
            throw new RuntimeException();
    }
    ResourceStore loadResourceStore(...) {
        checkClosed();
        StoreEntry se = lookupEntry(...);
        return se.getStore();
    }
    ...
}

synchronized Entry lookupEntry(...) {
    Entry e = (Entry) entries.get(...);
    if (e == null) {
        e = new Entry();
        entries.put(..., e);
    }
    return e;
}

synchronized void shutdown() {
    while (...) {
        // remove all entries
    }
    closed = true;
}

```

Fig. 4. Violation of atomicity in class `ResourceStoreManager` of the `jigsaw` benchmark.

The benchmarks `mol` and `ray` share part of the code and both report violation for lock views with disjoint variable sets on class `JGFTimer`. There is indeed a notion of consistency among the variables that could be violated if method calls would be interleaved in a particular sequence. There is however an explicit runtime check that detects this situation and issues a warning.

So far, views are restricted to shared variables. We have experimented with a further restriction: reads are only entered into lock views if the value is exposed outside the lock scope or method (i.e., the value is returned from a `synchronized`

method or assigned to a stack escaping object). This modification reduces the number of reports by around 30-50%, however some cases of high-level data races are not recognized any more.

5 Conclusions

Violations of atomicity at the level of methods are common in parallel programs; in some cases however, such violations are undesired and considered as synchronization faults. This work presents a practical automated approach to detect such violations through an efficient whole program analysis. The detection is based on a notion of consistency (method consistency) that is inferred from the usage of locks and the accesses to shared data. Violations of method consistency match common cases of critical atomicity violations. Hence, method consistency is useful to identify a certain class of synchronization defects.

The common use of concurrent programming languages makes static tools for the automated detection of synchronization defects increasingly important. The focus of our static analysis on objects that are potentially shared makes the analysis efficient. The narrowing of the consistency notion to variables of the same class and the compaction of method reports along the caller hierarchy result in a moderate number of reports for the programs we assessed. Most reports do not reflect actual program defects. However, some of those reports that do not reflect a fault in the context of the analyzed program shed light on a synchronization discipline that could be insufficient if the affected data structure is reused in a different context.

Acknowledgments

We thank Cyrille Artho for helpful discussions and Matteo Corti and Florian Schneider for their contributions to the compiler system.

References

1. C. Artho, K. Havelund, and A. Biere. High-level data races. In *Rec. Wkshp. Verificat. and Validat. of Enterprise Information Systems (VVEIS'03)*, Apr. 2003.
2. D. Bacon, J. Bloch, J. Bogda, C. Click, P. Haahr, D. Lea, T. May, J.-W. Maessen, J. Mitchell, K. Nielsen, and B. Pugh. The “Double-Checked Locking is Broken” Declaration. <http://www.cs.umd.edu/~pugh/java/memoryModel>, 2000.
3. C. Flanagan, K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proc. Conf. (PLDI'02)*, June 2002.
4. C. Flanagan and S. Quadeer. A type and effect system for atomicity. In *Proc. Conf (PLDI'03)*, June 2003.
5. C. Flanagan and S. Quadeer. Types for atomicity. In *Intl. Workshop on Types in Language Design and Implementation (TLDI'03)*, Jan. 2003.
6. R. Lipton. A method of proving properties of parallel programs. *Comm. of the ACM*, 18(12):717–721, Dec. 1975.
7. E. Ruf. Effective synchronization removal for Java. In *Proc. Conf. (PLDI'00)*, pages 208–218, June 2000.
8. C. von Praun and T. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *Proc. Conf. (PLDI'03)*, June 2003.
9. L. Wang and S. Stoller. Run-time analysis for atomicity. In *Rec. Workshop on Runtime Verification (RV'03)*, July 2003.

Checking Concise Specifications for Multithreaded Software

Stephen N. Freund¹ and Shaz Qadeer²

¹ Department of Computer Science, Williams College, Williamstown, MA 01267

² Microsoft Research, One Microsoft Way, Redmond, WA 98052

Abstract. Ensuring the reliability of multithreaded software systems is difficult due to the potential for subtle interactions between threads. Unfortunately, checking tools for such systems do not scale to programs with a large number of threads and procedures. To improve this shortcoming, we present a verification technique that uses concise specifications to analyze large multithreaded programs modularly. We achieve thread-modular analysis by annotating each shared variable by an *access predicate* that summarizes the condition under which a thread may access that variable. We achieve procedure-modular analysis by annotating each procedure by its specification, which is related to its implementation by an abstraction relation that combines the notions of *simulation* and *reduction*. We have implemented our analysis in Calvin-R, a static checker for multithreaded Java programs. To validate our methodology, we have used Calvin-R to check a number of important properties for a file system. Our experience shows that requirements for complex multithreaded systems can be stated concisely and verified in our framework.

1 Introduction

Software verification is an important and difficult problem. Over the past few decades, a variety of techniques based on dataflow analysis, theorem proving, and model checking have emerged for the analysis of sequential software. However, these techniques have not yet enabled verification of large, multithreaded software systems. Since concurrency is an insidious source of programming errors, multithreaded programs would benefit significantly from automated error-detection tools. The need for such tools will continue to grow as multithreaded software becomes even more widespread, expanding from the domain of low-level systems software (operating systems and databases) to most programs written in high-level languages like Java [2] and C# [12]. In this paper, we present a new modular verification technique for multithreaded Java programs.

Modularity is the key to scaling program analyses to large software systems. For sequential programs, modular analysis is achieved through pre- and post-conditions for procedures. However, due to interaction among the threads, pre- and post-conditions are insufficient for modular verification of multithreaded programs. Jones [22] proposed the first proof rule for modular verification of multithreaded programs. The proof rule of Jones required, in addition to pre- and post-conditions, a rely-guarantee specification for each procedure to capture the interaction among the threads. Both the rely and the guarantee specifications are actions (binary relations on the shared store). While the guarantee specification is a requirement on the updates performed by the thread executing the procedure, the rely specification is a requirement on the updates performed by the other threads.

In previous work [16], we extended Jones' method by generalizing the guarantee of a procedure from a single action to a program constructed from actions. This *guarantee program* has the property that every sequence of atomic updates to shared variables in the implementation is matched by a sequence of atomic updates in the specification. The implementation and guarantee of a procedure must be related via *simulation*. Simulation provides *data abstraction* for multithreaded programs by supporting abstract descriptions of the sequence of actions performed during the execution of a procedure. In particular, simulation allows steps that modify only local variables to be abstracted away.

This generalization is crucial for allowing modular specification and verification of a multithreaded library independently from the clients of the library.

Unfortunately, simulation requires every step in a procedure's implementation that updates a shared variable to be matched by a step in its specification. Consider, for example, a multithreaded program in which the shared variable `count` is protected by the mutex `m`. The following procedure increments `count` by one.

```
void increment() { acquire(m); int j = count; j++; count = j; release(m); }
```

A specification that simulates the implementation must have at least three steps corresponding to acquiring `m`, updating `count`, and releasing `m`. Consequently, such a specification is no more concise or intuitive than the implementation. Although the programmer's intuition is that the execution of `increment` by a thread appears to happen in "one step" (rather than three), simulation does not allow such a specification to be proved.

We introduce a new and more expressive criterion for relating the implementation of a procedure to its specification in this paper. The new relation augments simulation with the notion of reduction, which was first introduced by Lipton [24]. The notion of reduction is based on commuting operations performed by different threads when they do not interfere with each other. An operation that commutes to the right of a succeeding operation by a different thread is a *right mover*, and an operation that commutes to the left of a preceding operation by a different thread is a *left mover*. For example, the operation `acquire(m)` is a right mover, and the operation `release(m)` is a left mover. Moreover, since all threads access `x` only while holding the mutex `m`, the read (and write) operation to `count` is both a right and a left mover since no other thread can concurrently access `count`. Any execution sequence in which a thread performs a sequence of right movers followed by a single atomic operation followed by a sequence of left movers can be viewed as occurring "in one step". The execution of `increment` by a thread has this property. To check that `increment` atomically increments `x` by one, we first apply reduction and then check simulation only on the reduced sequence. Thus, reduction allows control abstraction for multithreaded programs akin to pre- and post-conditions for sequential programs.

Our experience with multithreaded software checking indicates that the most intuitive and concise specifications for procedures in multithreaded programs are obtained by combining simulation and reduction. In fact, the specification of a procedure in our framework is often no more complex than its specification would be under the assumption that the program is single-threaded. We also show in Section 2.2 that the ability to use both simulation and reduction yields verifiable specifications that can not be expressed with just reduction or just simulation. We are not aware of any other automated checking tool that uses a combination of simulation and reduction to check abstraction.

In order to apply reduction to code sequences that access shared variables, the locking discipline for these shared variables must be specified by the programmer. Although mutexes are the most common synchronization discipline, there are a variety of other mechanisms used in practice [28, 14]. To capture these idioms, we introduce access predicates, a novel and general mechanism for specifying a wide variety of synchronization mechanisms including mutexes, readers-writer locks, data-dependent locking, etc. The access predicate expresses the condition under which a thread may access that variable. Our verification technique checks the code of each thread assuming that the environment (containing other threads) behaves according to the access predicates.

We have implemented our analysis in the Calvin-R checker, an extension of the Calvin checker for multithreaded Java programs [14, 16]. Our tool modularly checks that each method in a program satisfies the access predicates and is abstracted by its specification. For each check, Calvin-R

constructs a sequential program capturing the necessary correctness requirements and verifies that it does not go wrong using existing verification techniques for sequential programs. Specifically, we employ verification conditions [10, 17] and the Simplify automatic theorem prover [26].

To validate our approach, we have used Calvin-R to check many properties of Daisy, a simple NFS file server we designed as our first case study. Daisy uses synchronization mechanisms similar in complexity to those found in other file systems. We have verified that all procedures in Daisy satisfy the access predicates, showing that the code adheres to the specified synchronization discipline. In addition, we specified and checked functional requirements on a number of the most complex procedures in Daisy. Our notion of abstraction invariably led to intuitive and concise specifications, and we uncovered several unknown bugs, primarily in the code for handling errors.

We present an overview of our verification technique in Section 2 through several examples. Due to space limitations, we omit the full formal presentation of our modular analysis and soundness proof. The full details may be found in our companion technical report [18]. We present a discussion of the related work in Section 3 and conclude in Section 4.

2 Verification Technique

To demonstrate our specification and verification system, we present two example programs—a class that implements a counter, and a class that implements block allocation in a simple file system. In both cases, we state concise specifications for the code and describe how our analysis checks them.

2.1 Counter

The following class implements a counter:

Figure 1: Counter

```
class Counter {
  int m /*@ accessible_if m == 0 || m == tid */;
  int count /*@ accessible_if m == 0 || m == tid */;

  /*@ performs action (count) { \old(m) == 0 && count == \old(count) + 1 } */
  void increment() {
    acquire(m); int j = count; j++; count = j; release(m);
  }
}
```

The `increment` method adds one to `count`. Concurrent calls to `increment` are serialized using the mutex lock `m`, which is acquired at the beginning and released at the end of `increment`. We model the mutex as an integer variable whose value is the identifier of the thread holding the mutex, or 0 if it is not held. The atomic operation `acquire(m)` blocks until `m` is 0 and then sets `m` to the identifier of the currently executing thread (`tid`), and the atomic operation `release(m)` sets `m` back to 0.

The `performs` annotation specifies method behavior. According to its specification, the `increment` method behaves as if at some point during its execution, it performs a single atomic action that (1) modifies only the variable `count`, which is indicated with the `modifies` clause `(count)`; and (2) blocks until the value of `m` is 0 and then increments `count` by 1. The values `\old(m)` and `\old(count)` refer to the variable values in the pre-state of this action.

The `accessible_if` annotations indicate to our checker the access predicates for the variables. The access predicates for `m` and `count`, denoted A_m and A_{count} , express the requirement that a

thread t may access $\mathbf{m}$ and `count` only if $\mathbf{m} = 0$ or $\mathbf{m} = t$. It is worth noting that the access predicates for the shared variables do not preclude data races. As we have pointed out earlier [15], absence of data races is neither necessary nor sufficient for atomicity. Hence, our analysis allows general `accessible_if` predicates for shared variables.

The method `increment` can be called (possibly concurrently) by any number of threads. To ensure the specification of `increment` is valid for any calling thread, Calvin-R checks the method for an arbitrary thread t with the assumption that other threads operating concurrently with thread t access $\mathbf{m}$ and `count` according to the access predicates. For each execution trace of the method in such an environment, the tool checks both that thread t satisfies the access predicates and that the execution is abstracted by an execution of the specification of `increment` in the same environment. The first check can be easily performed using the technique of thread-modular verification [22, 14, 16].

To check abstraction, Calvin-R *reduces* the given execution to another execution that ends in the same final store and in which the operations performed by thread t (in `increment`) happen atomically without any interleaved actions by the environment. After reduction, the tool checks that the single atomic action in the specification of `increment` *simulates* the composition of the consecutively occurring actions of thread t .

To reduce an execution trace, the tool shows that the operations by thread t in the execution form a sequence of zero or more right-commuting operations (*right movers*) followed by a single operation followed by a sequence of zero or more left-commuting operations (*left movers*).

An operation of t is a right mover if, immediately after its execution, no other thread can access a variable accessed by the operation. Since the environment operations behave according to the access predicates, we can derive the condition to verify that an operation is right-mover from the access predicates. For example, if the operation by thread t accesses variable $\mathbf{m}$, the condition $E_{\mathbf{m}}(t)$ must be shown in the post-store:

$$\begin{aligned} E_{\mathbf{m}}(t) &= \forall j \in Tid. j \neq t \Rightarrow \neg A_{\mathbf{m}}(j) \\ &= \forall j \in Tid. j \neq t \Rightarrow \neg(\mathbf{m} = 0 \vee \mathbf{m} = j) \\ &= (\mathbf{m} = t) \end{aligned}$$

Thus, to prove that an operation by thread t accessing $\mathbf{m}$ is a right mover, we must show that thread t holds $\mathbf{m}$ in the post-store of the operation. Intuitively, the predicate $E_{\mathbf{m}}(t)$ is the condition under which thread t has *exclusive access* to $\mathbf{m}$.¹ The condition for an operation accessing `count` is identical. We may commute a right mover with that operation following it because we are guaranteed that the two operations access disjoint sets of variables.

Similarly, an operation is a left mover if we can prove that $\mathbf{m}$ is held by thread t in the pre-store. For `increment`, we can show that `acquire(m)` and all subsequent operations until `release(m)` are right movers and `release(m)` is a left mover. Therefore, the code in `increment` is reducible to a single action.

Figure 2(a) shows an execution of `increment` with arbitrary actions of other threads and interleaved between the steps of t . Actions from other threads are labeled **X**, and for simplicity we insert only a single such action between consecutive actions of t . Execution (b) shows the reduced execution.

¹ Alternatively, we could have specified the exclusive access predicate and derived the access predicate from it.

Figure 2: Checking abstraction for increment

The simulation check is straight-forward once the execution has been reduced, as shown in (c).

Note that the specification of `increment` shown above is comparable in complexity to the post-condition specification for `seq_increment`, a version of `increment` designed for sequential programs:

```
/*@ modifies count; ensures count == \old(count) + 1 */
void seq_increment() { int t = count; t++; count = t; }
```

2.2 Block allocation in Daisy

To further illustrate the importance of using both reduction and simulation for proving succinct procedure specifications, we present in Figure 3 the code for block allocation from the Daisy file system. Daisy is a simple NFS file system we designed as our first case study for Calvin-R. The file system synchronization mechanisms are similar in complexity to those found in other file systems. However, the data structures and algorithms in Daisy are relatively simple, allowing us to implement it in approximately 1200 lines of Java code.

The `alloc` function searches for a free file system block by finding a false bit in the `bits` array. The flag `bits[j]` indicates whether the j -th disk block is currently in use. When `alloc` identifies a free block, it allocates the block by setting the appropriate bit to true and returns the index of the block with the lock corresponding to it still held. If `alloc` fails to find a free block, it returns -1. The `free` function takes a block index i as an argument and requires that the the mutex `l[i]` be held on entry to the function. It frees block i by setting `bits[i]` to false and returns after releasing `l[i]`.

The mutex `l[j]` guards the bit `bits[j]`. The `accessible_if` annotation on `bits` is parameterized by j to indicate this relationship for all j . A program acquires and releases lock `l[j]` by calling `acquire(l[j])` and `release(l[j])`, respectively. Locking at such a fine granularity is a standard technique for improving throughput in commercial file systems. However, it is also a major source of errors and demands substantial debugging effort. We are not aware of any other static tool for checking synchronization at such fine granularity.

The `performs` annotation for `alloc` is intuitive and mirrors the two possible outcomes of executing `alloc`. The specification is a choice between two atomic actions. In the first action, no free block is found and -1 is returned. The special variable `\result` refers to the value returned by a function.

Figure 3: Allocator

```

class Allocator {
    static int l[NBLOCKS] /*@ accessible_if[j] l[j] == 0 || l[j] == tid */;
    static boolean bits[NBLOCKS] /*@ accessible_if[j] l[j] == 0 || l[j] == tid */;

    /* performs action "NoBlocks": () { \result == -1 }
       [] action "Allocated": (bits[\result], l[\result]) {
           \old(l[\result]) == 0 && l[\result] == tid
           !\old(bits[\result]) && bits[\result]
       }
    */
    public static int alloc() {
        for (int i = 0; i < NBLOCKS; i++) {
            acquire(l[i]);
            if (!bits[i]) {
                bits[i] = true;
                /* witness "Allocated";
                return i;
            }
            release(l[i]);
        }
        /* witness "NoBlocks";
        return -1;
    }

    /* requires l[i] == tid
       performs action "Free": (bits[i], l[i]) { l[\result] == 0 && !bits[\result] }
    */
    public static void free(int i) {
        bits[i] = false;
        release(l[i]);
        /* witness "Free";
    }
}

```

In the second action, the return value is the index of an unused block. This action blocks until the mutex protecting the allocation bit of the block is zero, and it then updates the bit from false to true. The *witness* annotations in the code indicate program points where simulation steps in the specification may occur. We use the explicit witness to guide the simulation check by indicating the “commit points” in the implementation of the atomic actions in the specification.² The specification for the *free* function is similar.

As in the *increment* example, Calvin-R checks that the implementation of *alloc* is abstracted by its specification for an arbitrary thread *t* in an environment that respects the access predicates. However, the checking of *alloc* is significantly more complicated than that of *increment*. The *alloc* function has an unbounded number of execution sequences, each consisting of 0 or more *acquire-test-release* subsequences, followed by either a return of -1 or an *acquire-test-set* and a return of a non-negative index. Such executions are not reducible to a single atomic action. Therefore, our checker decomposes the sequence of actions performed by thread *t* into subsequences that are each reducible to a single action, as shown in Figure 4(a) and (b) for one possible execution.

Calvin-R deduces that each of the *acquire-test-release* subsequences is reducible to a single atomic action, and further checks that each of these actions is simulated by *skip*, an action that leaves every variable unchanged. If there is no final *acquire-test-set* sequence, then Calvin-R further deduces that

² In general, finding the correspondence between implementation steps and specification steps is a hard problem.

Figure 4: Checking abstraction for alloc

the last implementation action returns -1 and is therefore simulated by the action "NoBlocks" of the specification. If there is a final *acquire-test-set* sequence followed by the return of a non-negative index, Calvin-R reduces it to a single atomic action and checks that it is simulated by the action "Allocated" of the specification. In both cases, by first using reduction and then simulation, Calvin-R abstracts the execution to a (possibly empty) sequence of `skip` action followed by an action from the specification.³ Figure 4 illustrates one possible execution of `alloc`. Execution (b) shows the reduced execution of (a), and (c) and (d) demonstrate the simulation. We divide the simulation into two steps to show that simulation involves composing a sequence of actions into a single action, as well as generalizing an action.

Although `alloc` uses fine-grained synchronization, our method allows us to prove a concise and intuitive specification that is similar in complexity to the specification of `alloc` assuming single-threaded execution.

3 Related work

Lipton [24] first proposed reduction as a way to reason about concurrent programs without considering all possible interleavings. He focused primarily on checking deadlock freedom. Doepfner [11], Back [3], and Lamport and Schneider [23] extended this work to allow proofs of general safety properties. Misra [25] has proposed a reduction theorem for programs built with monitors [21] communicating via procedure calls. Cohen and Lamport [8] have extended reduction to allow proofs of liveness properties. All of these papers have focused on the theory of reduction. However, they do not describe a methodology for verifying programs. We go beyond their work by developing a specification and verification methodology for a widely used programming language.

Partial-order methods [20, 27] have been used to limit state-space explosion while model checking concurrent programs. These methods identify sequences of interleaved steps for which the property being checked is insensitive to the exact ordering. A single representative interleaving of the operations is then explored. These methods have mostly been applied to systems built from processes communicating through message passing. Verisoft [19] is an example of such a tool. While these methods are typically unable to reorder accesses to shared variables, Calvin-R uses access predicates to determine when it is safe to reorder accesses to shared variables as well.

Using ideas from reduction and partial-order methods, Bruening [5] has built an assertion checker based on state-space exploration for multithreaded Java programs. His tool requires another checker to ensure the absence of races. This assumption allows `synchronized` code blocks to be treated as atomic. Stoller [31] provides a generalization of Verisoft and Bruening's method to allow model checking of programs with either message-passing or shared-memory communication. Both of these approaches are restricted to mutex-based synchronization and operate on the concrete program without any abstraction. In our work, access predicates provide a more general mechanism for specifying synchronization. More recently, Stoller and Cohen have adopted access predicates in order to capture a richer set of synchronization idioms and to perform reduction during model checking [32].

Flanagan and Qadeer use reduction in a type system to identify procedures in multithreaded programs that may be considered to execute atomically [15]. Their syntactic type-based analysis is more scalable because it requires fewer and less complex program annotations, but it is limited to

³ The `performs` specification implicitly allows arbitrary number of `skip` actions at any control point.

checking this single atomicity property. In contrast, our paper presents a modular semantic analysis based on verification conditions and theorem proving that can check many complex properties of multithreaded programs.

A number of static tools have been designed to detect data races. These include several type systems [13, 4], Warlock [30], and a race detector for SPMD programs [1]. Dynamic race detection tools [29, 6] require very few annotations, if any, but may fail to detect some errors due to insufficient coverage. Several tools combining dynamic and static analyses have recently been proposed [33, 7]. Access predicates provide a simple, general method for specifying and verifying a wider variety of synchronization mechanisms than allowed by these tools.

Several tools [9, 34] verify safety properties using a combination of data abstraction and model checking. These tools consider all possible thread interleavings while performing state exploration. The approach in this paper can be used to abstract a program, thereby reducing the possible interleavings. Invariant checking can then be performed on the abstract program thus improving the efficiency of these techniques.

4 Conclusions

Enforcing program specifications statically can greatly improve software quality. However, the inability to express complex properties in a simple, intuitive way is a major impediment to adopting many specification-based program checking tools, especially those targeting multithreaded code.

We outline in this paper a checking methodology that simplifies the task of specifying and verifying multithreaded programs. We allow thread synchronization mechanisms to be concisely expressed as access predicates, and we also allow concise, yet expressive specifications for procedures. The specification of a procedure is related to its implementation by a powerful abstraction relation that allows both data and control abstraction by combining the theories of simulation and reduction. Our access predicates and procedure specifications enable program analysis that is both thread-modular and procedure-modular. We have implemented our analysis in the Calvin-R checker for multithreaded Java programs and checked important properties of multithreaded systems with this tool.

The next step is to validate this methodology further by showing that it can scale to larger programs. Some issues to address are how to best map errors in the generated sequential program back to errors in the original, multithreaded program; how to reduce annotation overhead by automatically inferring some annotations, such as simple access predicates for variables guarded by mutual exclusion locks; and how to ensure that our translation does not produce sequential programs too complex for the underlying theorem prover to handle.

Acknowledgments: We thank Cormac Flanagan for the initial design of Daisy and for useful discussions and feedback on this paper. We also thank Sanjit Seshia for useful discussions, and Mark Lillibridge for comments on a draft of this paper.

References

1. A. Aiken and D. Gay. Barrier inference. In *Proceedings of the 25th Symposium on Principles of Programming Languages*, pages 243–354, 1998.
2. K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.
3. R.-J. Back. A method for refining atomicity in parallel algorithms. In *PARLE 89: Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science 366, pages 199–216. Springer-Verlag, 1989.

4. C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Proceedings of 16th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 56–69, October 2001.
5. D. Bruening. Systematic testing of multithreaded java programs. Master's thesis, Massachusetts Institute of Technology, 1999.
6. G. Cheng, M. Feng, C. Leiserson, K. Randall, and A. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the 10th annual ACM Symposium on Parallel Algorithms and Architectures*, pages 298–309, 1998.
7. J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 258–269, 2002.
8. E. Cohen and L. Lamport. Reduction in TLA. In *International Conference on Concurrency Theory*, pages 317–331, 1998.
9. J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.
10. E. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
11. T. Doeppner, Jr. Parallel program correctness through refinement. In *Proceedings of the 4th Symposium on Principles of Programming Languages*, pages 155–169, 1977.
12. ECMA. Standard ECMA-334: C# Language Specification, 2002. Available on the web as <http://www.ecma-international.org/publications/files/ecma-st/Ecma-334.pdf>.
13. C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 219–232, 2000.
14. C. Flanagan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *Proceedings of European Symposium on Programming*, pages 262–277, 2002.
15. C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation*, 2003.
16. C. Flanagan, S. Qadeer, and S. Seshia. A modular checker for multithreaded programs. In *CAV 02: Computer Aided Verification*, pages 180–194, 2002.
17. C. Flanagan and J. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the 28th Symposium on Principles of Programming Languages*, pages 193–205. ACM, 2001.
18. S. N. Freund and S. Qadeer. Checking concise specifications of multithreaded software. Williams College Technical Note 01-2002 (available from <http://www.cs.williams.edu/~freund/>), 2002.
19. P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th Symposium on Principles of Programming Languages*, pages 174–186, 1997.
20. P. Godefroid and P. Wolper. A partial approach to model checking. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science*, pages 406–415. IEEE Computer Society Press, 1991.
21. C. Hoare. Monitors: an operating systems structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.
22. C. Jones. Specification and design of (parallel) programs. In R. Mason, editor, *Information Processing*, pages 321–332. Elsevier Science Publishers B. V. (North-Holland), 1983.
23. L. Lamport and F. Schneider. Pretending atomicity. Research Report 44, DEC Systems Research Center, 1989.
24. R. Lipton. Reduction: A method of proving properties of parallel programs. In *Communications of the ACM*, volume 18:12, pages 717–721, 1975.
25. J. Misra. *A Discipline of Multiprogramming: Programming Theory for Distributed Applications*. Springer-Verlag, 2001.
26. C. Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox Palo Alto Research Center, 1981.
27. D. Peled. Combining partial order reductions with on-the-fly model checking. In D. Dill, editor, *CAV 94: Computer Aided Verification*, Lecture Notes in Computer Science 818, pages 377–390. Springer-Verlag, 1994.
28. M. Rinard. Analysis of multithreaded programs. In *SAS 01: Static Analysis Symposium*, Lecture Notes in Computer Science 2126, pages 1–19. Springer-Verlag, 2001.
29. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.
30. N. Sterling. WARLOCK — a static data race analysis tool. In *USENIX Technical Conference Proceedings*, pages 97–106, Winter 1993.
31. S. D. Stoller. Model-checking multi-threaded distributed java programs. In *Proceedings of the 7th International SPIN Workshop on Model Checking and Software Verification*, Lecture Notes in Computer Science 1885, pages 224–244. Springer-Verlag, 2000.
32. S. D. Stoller and E. Cohen. Optimistic synchronization-based state-space reduction. In H. Garavel and J. Hatcliff, editors, *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2619 of *Lecture Notes in Computer Science*, pages 489–504. Springer-Verlag, 2003.
33. C. von Praun and T. Gross. Object-race detection. In *16th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 78–82, 2001.
34. E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Proceedings of the 28th Symposium on Principles of Programming Languages*, pages 27–40, 2001.

Stronger Typings for Separate Compilation of Java-like Languages (Extended Abstract)*

Davide Ancona and Giovanni Lagorio

DISI - Università di Genova
Via Dodecaneso, 35, 16146 Genova (Italy)
email: {davide,lagorio}@disi.unige.it

Abstract. We define a formal system supporting separate compilation for a small but significant Java-like language.

This system is proved to be stronger than the standard compilation of both Java and C#, in the sense that it better supports software reuse by avoiding unnecessary recompilation steps after code modification which are usually performed by using the standard compilers.

This is achieved by introducing the notion of *local type assumption* allowing the user to specify weaker requirements on the source fragments which need to be compiled in isolation.

Another important property satisfied by our system is compositionality, which corresponds to the intuition that if a set of fragments can be separately compiled and such fragments are compatible, then it is possible to compile all the fragments together as a unique program and obtain the same result.

1 Introduction

Separate compilation of statically typed languages is an important feature of modern systems, since it promotes software reuse while retaining type safety and semantic consistency of programs.

As pointed out by Cardelli [7], separate compilation essentially corresponds to a typing judgment which has the general form $\Gamma \vdash S:\tau$, where Γ is a type environment containing all necessary information needed for compiling in isolation the source fragment S , and τ is the inferred type of S . Using the terminology introduced in [7], we say that fragment S *intrachecks* in Γ and has type τ .

For instance, in the SDK systems, that is, those specifying the SDK Java¹ compiler [3, 9, 12], S corresponds to the declaration of a class C , Γ_s is a *standard* type environment containing all the information on the classes needed by C and having the general form $C_1:\tau_1, \dots, C_n:\tau_n$, and τ and τ_i , for $i = 1..n$, are *class*

* Partially supported by Dynamic Assembly, Reconfiguration Type-checking - EC project IST-2001-33477, APPSEM II - Thematic network IST-2001-38957, and Murst NAPOLI - Network Aware Programming: Oggetti, Linguaggi, Implementazioni.

¹ Throughout this paper we will only mention Java for brevity, but all claims about Java can be replaced with analogous claims about C#.

types obtained by extracting all type annotations in the code, that is, the direct superclass, the headers of methods and so forth.

Another crucial notion for separate compilation is *interchecking* [7]; in general, the fact that two fragments S_1 and S_2 , named C_1 and C_2 , respectively, intracheck does not guarantee that the program obtained by merging S_1 with S_2 is statically correct. To ensure this we need to either recompile S_1 and S_2 as a whole² (but this solution obviously contradicts separate compilation), or just verify that S_1 and S_2 intercheck by proving that the type environment $\Gamma_s = C_1:\tau_1, C_2:\tau_2$ is *stronger* than both the environments Γ_1 and Γ_2 used for intrachecking S_1 and S_2 , respectively.

In order to have an effective interchecking process, the notion of “stronger type environment” need to be syntactically captured by an *entailment* relation $\vdash$ between type environments, which must be at least sound: $\Gamma_1 \vdash \Gamma_2$ implies that Γ_1 is stronger than Γ_2 , that is, for all S and τ , if $\Gamma_2 \vdash S:\tau$ then $\Gamma_1 \vdash S:\tau$. In the simplest cases the entailment relation reduces to type environment inclusion, but is not so simple in the system presented here.

The notion of interchecking can be fruitfully used for enhancing *selective recompilation* [1], which is the ability of avoiding unnecessary recompilation steps after code modification, while retaining both type safety and semantic consistency of programs. As an example, let us consider the following two classes, assuming that they are defined in separate fragments:

```
class H extends P {
    int g(P p) {return p.f(new H());}
}
class P extends Object{
    int f(Object o){return 1;}
}
```

Classes H and P both intracheck and intercheck. Now let us change the definition of P ; obviously P needs to be recompiled (let us assume that such a recompilation is successful), but what about H ? In Java three different cases may occur:

Case 1. The modification of P invalidates neither the type safety nor the semantic consistency of H : if we recompile H , then we get no error and the same bytecode. This happens, for instance, if we add to P a new method `int h()`.

Case 2. The modification of P does not invalidate the type safety, but does compromise the semantic consistency of H : if we recompile H we get no error but we obtain a different bytecode. For instance, if the parameter type of `f` becomes P , then the bytecode for H must change correspondingly, since in the Java bytecode method calls are annotated with the types of the parameters of the resolved method.

Case 3. The modification of P compromises the type safety of H : if we recompile H we get an error; for instance, this happens if we change into `int` the parameter type of `f`.

In case 1 we would like to avoid an unnecessary recompilation of class H , while in case 2 we do want to force recompilation of H ; however, this is not possi-

² For simplicity, we only consider interchecking of closed programs, even though this notion could be easily extended to open programs.

ble with the SDK compiler³, where the user can choose either to recompile all classes (avoiding type unsafety and semantic inconsistency, but not unnecessary recompilation), or to recompile only modified classes (avoiding unnecessary recompilation, but not type unsafety and semantic inconsistency). For what concerns case 3, the best solution would consist in detecting the problem without reinspecting the source code for H, but since this task turns out to be quite hard, in the system presented in this paper cases 2 and 3 cannot be distinguished. From the example above we can draw the following important conclusions:

- Case 2 shows that some class modification can affect the bytecode of other unchanged classes, because of the type annotations in the bytecode needed for run-time resolution of methods and verification of classes [14, 10]. Therefore, in order to capture case 2, the intrachecking judgment needs to be extended to take into account the generated code. In this paper we use an intrachecking judgment of the form $\Gamma \vdash S : \tau \rightsquigarrow B$ with the meaning “in the type environment Γ the source fragment S intrachecks, has type τ , and compiles to the binary fragment B ”.
- In order to avoid an unnecessary recompilation in case 1, we can just redo interchecking for class H by using the entailment relation between type environments. Let Γ_H denote the type environment used for intrachecking class H before P was modified, so that $\Gamma_H \vdash S_H : \tau_H \rightsquigarrow B_H$ (for the appropriate S_H , τ_H and B_H), and let τ'_P denote the new type of P after its modification. If we can prove $H:\tau_H, P:\tau'_P \vdash \Gamma_H$, then we can conclude that $H:\tau_H, P:\tau'_P \vdash S_H : \tau_H \rightsquigarrow B_H$ holds, and, therefore, recompilation of H is unnecessary.
- In order to minimize the number of unnecessary recompilations, intrachecking typings should be as *strong* as possible. Adapting the formal definitions given by Wells [18] to our intrachecking judgments, a *typing* t is a pair $\langle \Gamma, \tau \rangle$ and $\text{Terms}(t)$ denotes the set of all pairs $\langle S, B \rangle$ s.t. $\Gamma \vdash S : \tau \rightsquigarrow B$ is provable in the system. Then, a typing t_1 is *stronger* than a typing t_2 if and only if $\text{Terms}(t_1) \subseteq \text{Terms}(t_2)$ (*strictly stronger* when the inclusion is proper). Intuitively, if t_1 is stronger than t_2 and $S \in \text{Terms}(t_1)$, then t_1 approximates S better than t_2 and, therefore, t_1 is preferable to t_2 since it enlarges the set of contexts where S can be used in a type safe and semantically consistent way without recompiling it.

Ideally, the best situation would be a type system with principal typings, where for each correct fragment there would exist the strongest typing [18]. However, in this paper we do not propose to solve the problem of principality for Java-like languages. Instead, we formally investigate a system where typings are strictly stronger than those of the SDK systems. The proof that our system has principal typings can be found in [5].

Indeed, as already argued in a previous paper [4], the intrachecking typings $\langle \Gamma, \tau \rangle$ in the SDK systems are too weak. For instance, in order to successfully compile class H, the SDK compiler retrieves all the type information on class P,

³ Throughout the paper we refer to the compiler of Java 2 SDK, version 1.4.1.

including its direct superclass, its method headers, and so on; therefore, class `H` is intrachecked under the assumptions that class `P` extends `Object` and its body exactly contains just one method, named `f`, with one parameter of type `Object`, and return type `int`. However, these type assumptions for intrachecking `H` are far from minimal. We would like to be able to express less restrictive assumptions like, for instance, “the invocation `p.f(new H())` can be resolved to a method `f`, with parameter type `Object`, and return type `int`”.

While in functional programming, typings are made stronger by extending systems with more accurate types (for instance, intersection types) [18], here we take a dual approach by extending the SDK systems with “more accurate” type environments, that is, type environments able to express weaker requirements on classes. This is due to the fact that some Java features, like method overloading, require global analysis and thus conflict with modularity [7], therefore standard type environments assigning types to single classes are not expressive enough.

We extend standard type environments with local type assumptions [4], e.g., $C1 \leq C2$, which requires class `C1` to be a subtype of `C2`, but says nothing, for instance, about the methods of `C1`; however, for Java other kinds of local type assumptions are needed.

In this paper we focus on two related issues. First, the system presented here is an evolution of the system defined in a previous paper [4], where two important properties are proved: *compositionality*, which does not hold for our previous system, and the fact that typings are strictly stronger than those of the SDK systems. Second, we show how this system can be effectively used for enhancing Java selective recompilation [8].

Compositionality is an expected property of separate compilation defined by Cardelli [7] as follows: “The linked program should have the same effect as a program obtained by merging all the sources together and compiling the result in a single step”. In our system this amounts to requiring that if $\Gamma_1 \vdash S_1 : \tau_1 \rightsquigarrow B_1$, $\Gamma_2 \vdash S_2 : \tau_2 \rightsquigarrow B_2$ and S_1 and S_2 intercheck, then the program $\langle S_1 S_2 \rangle$ obtained by putting together the two fragments compiles successfully in the empty environment and produces the pair of binaries $\langle B_1 B_2 \rangle$.

Another interesting property is that our system is effectively “stronger” than the SDK ones: we prove that if Γ_s is a standard type environment, and $\Gamma_s \vdash S : \tau \rightsquigarrow B$, then there exists a strictly weaker type environment Γ which yields a stronger typing $\langle \Gamma, \tau \rangle$ for S and B , and, more interestingly, can be effectively constructed by collecting all local assumptions needed to prove $\Gamma_s \vdash S : \tau \rightsquigarrow B$.

This last result shows that, during the compilation of a closed program `P` (that is, a self-contained set of code fragments) it is possible to infer for each fragment of `P` the set of local type assumptions (that is, a type environment) which is really needed by the compiler for that particular fragment; the compiler can take advantage of these automatically generated type environments for enhancing Java selective recompilation. In particular, we show how this idea can be used for enhancing *Javamake*, the only Java-specific make technology we are aware of (at least in form of a publication) [8, 13].

The rest of the paper is structured as follows. Section 2 is a gentle introduction to the system, whereas Section 3 discusses the most important related work [8] and shows how our system can be used in practice for enhancing *Javamake*. Finally, Section 4 contains pointers to other related work and some conclusions. All the formal definitions and results can be found in the extended version of this paper [2].

2 An Informal Presentation

This section is a gentle introduction to the system formally defined in the extended version of this paper [2]. More precisely, the two basic notions of *local* type assumption and *entailment* relation between type environments are informally presented and motivated.

Local Type Assumptions Let us consider a little bit more involved version of the class H mentioned in the Introduction:

```
class H extends P {
  int g(P p) {return p.f(new H());}
  int m() {return new H().g(new P());}
  U id(U u){return u;}
  X em(Y y){return y;}
}
```

and let us analyze under which assumptions class H can be successfully compiled. If we take the approach of the SDK compiler, then we would need to impose rather strong requirements on the classes used by H, by asking for the most detailed type information about such classes.

In our system this corresponds to compile H in a type environment Γ_s which contains standard type assumptions on the classes P, U, X and Y. For instance, if Γ_s is defined by:

$$\Gamma_s = P:\langle \text{Object}, \text{int f(Object)} \rangle, U:\langle \text{Object}, \rangle, Y:\langle X, \rangle, X:\langle \text{Object}, \rangle$$

then we are assuming that class P extends **Object** and declares only `int f(Object)`, classes U and X both extend **Object** and are empty, and class Y extends X and is empty. An environment like Γ_s containing standard type assumptions only is called a *standard type environment*.

Under the assumptions contained in Γ_s class H can be successfully compiled to the following binary fragment B_H :

```
class H extends P {
  int g(P p) {return p.<<P.f(Object)int>>(new H());}
  int m() {return new H().<<H.g(P)int>>(new P());}
  U id(U u){return u;}
  X em(Y y){return y;}
}
```

Note that in our system a binary fragment is just like a source fragment except that invocations contain a *symbolic reference* $\ll C.m(T_1 \dots T_n)T \gg$ to a method, giving the name m , the parameter types $T_1 \dots T_n$ and the return type T of the method, as well as the class C in which the method is to be found (see [14] 5.1). Indeed, from our perspective the most critical difference between source and binary fragments is type annotations in the method invocations, since it makes the problem of separate compilation (that is, separate typechecking plus code generation) substantially different from that of separate typechecking, as already pointed out in the Introduction.

Let us now try to relax the strong assumptions in Γ_s by seeking an environment Γ_l containing other kinds of type assumptions which still guarantee that H compiles to the same binary fragment B_h , but impose fairly weaker requirements on classes P , U , X and Y .

A first basic request is that the compilation environment containing H must provide a definition for the four classes which H depends on. In our system this is expressed by a local assumption of the form $\exists C$, therefore Γ_l will contain at least the assumptions $\exists P, \exists U, \exists X, \exists Y$.

Let us now focus on each single class used by H .

Class P : in order to correctly compile class H (into B_h) the following additional assumptions on class P must be added to Γ_l :

- $P \not\leq H$: P cannot be a subtype of H since inheritance cannot be cyclic.
- $P \odot \text{int } g(P)$: P can be correctly extended with method $\text{int } g(P)$; indeed, according to Java rules on method overriding, if P has a method $g(P)$, then g must have the same return type int as declared in H . Analogous requirements are needed for the other methods declared in H .
- $P.f(H) \xrightarrow{\text{res}} \langle \text{Object}, \text{int} \rangle$: invocation of method f , for an object of type P and with an argument of type H , is successfully resolved to a method with a parameter of type Object and return type int . This assumption ensures that the body of g in H is successfully compiled to the same bytecode of method g in B_h (in other words, the same symbolic reference to the method is generated). Note that we do not need to know the class where the method is declared, since the bytecode is annotated with the type of the receiver.

Class U : no additional requirements on U are needed, since the static correctness of method id in H only requires the existence of U .

Classes X and Y : in order to correctly compile class H , class Y must be a subtype of class X , otherwise method em in H would not be statically correct. Therefore we need to add the assumption $Y \leq X$.

In conclusion, class H can be successfully compiled and produce B_h in the environment Γ_l defined by:

$$\Gamma_l = \exists P, \exists U, \exists X, \exists Y, P \not\leq H, Y \leq X, P \odot \text{int } g(P), \\ P \odot \text{int } m(), P \odot U \text{ id}(U), P \odot X \text{ em}(Y), P.f(H) \xrightarrow{\text{res}} \langle \text{Object}, \text{int} \rangle$$

Furthermore, Γ_l is weaker than Γ_s ; for instance, class U must extend Object and be empty in Γ_s , while in Γ_l it can extend any class and declare any method. The

notion of stronger type environment is syntactically captured by an entailment relation on type environments.

Entailment of Type Environments Referring to the previous example, in our system the fact that Γ_l is weaker than Γ_s is formalized by the following property: for all S, τ, B if $\Gamma_l \vdash S : \tau \rightsquigarrow B$ is provable, then $\Gamma_s \vdash S : \tau \rightsquigarrow B$ is provable as well. However, since the definition above cannot be directly checked in an effective way, the notion of stronger type environment needs to be captured by an *entailment* relation (that is, a computable relation) between type environments.

For instance, in our system $\Gamma_s \vdash \Gamma_l$ can be proved. Furthermore, the entailment relation is proved to be *sound*, that is, if $\Gamma_1 \vdash \Gamma_2$ can be proved and Γ_1 is consistent (in the sense that it does not contain contradictory assumptions⁴), then Γ_1 is stronger than Γ_2 . In the particular example, we can go further, by showing that Γ_l is actually strictly weaker than Γ_s .

Let us add in **H** the new method `int one(){return 1;}`. After this change, the new code for class **H** still intrachecks in Γ_s , whereas intrachecking of the same code in Γ_l fails, otherwise the system would not be compositional. To see this, let us consider the following new declaration for class **P**:

```
class P extends Object{
  int f(Object o){return 1;}
  P one(){return new P();}
}
```

The reader can easily verify that each type assumption in Γ_l about **P** is satisfied by the new version of **P** above, however if we put all classes together we obtain a statically incorrect program, since method `one` is redefined in **H** with a different return type. Therefore Γ_s is strictly stronger than Γ_l ; from this last claim and from the soundness of the entailment we can deduce $\Gamma_l \not\vdash \Gamma_s$.

Finally, we end this section with another example of provable entailment, by showing that Γ_l contains redundant assumptions which, in fact, can be removed without affecting the outcome of the compilation of class **H**.

Let us consider the type environment Γ'_l obtained from Γ_l by removing the two assumptions $\exists P$ and $\exists Y$. Then, both the entailments $\Gamma_l \vdash \Gamma'_l$ and $\Gamma'_l \vdash \Gamma_l$ can be proved, hence Γ_l and Γ'_l are equivalent. The first entailment is trivial to prove, since Γ'_l is included in Γ_l ; the proof of the second entailment relies on the validity of the following two entailments:

$$P \odot \text{int } g(P) \vdash \exists P \qquad Y \leq X \vdash \exists Y$$

Intuitively, these two entailments must be provable because assumptions $P \odot \text{int } g(P)$ and $Y \leq X$ can be verified only in presence of a definition for **P** and **Y**, respectively. On the other hand, $\exists X$ cannot be entailed from $Y \leq X$; to see this, let us consider, for instance, the program fragment `class Y extends X {}`: it verifies $Y \leq X$ and $\exists Y$, but not $\exists X$.

⁴ Consistency can be checked by a polynomial time algorithm [2].

3 Selective Recompileation

While several papers have been written on the subject of selective recompilation (see Section 4), to our best knowledge only Dmitriev [8] and Lagorio [13] has focused on Java. Dmitriev’s paper describes a make technology, based on smart dependency checking, that aims to keep a project (that is, a set of source and binary fragments) consistent while reducing the number of files to be recompiled. A project is said to be consistent when all its sources can be recompiled producing the same binaries as before. The main idea is to catalog all possible changes to a source code (as, for instance, adding/removing methods) establishing a criterion for finding a subset of dependent classes that have to be recompiled. A freely downloadable tool, *Javamake*, is based on such a paper and implements the selective recompilation for Java upon any Java compiler. This tool stores some type information for each project in database files which are used to determine which changes have been made to the sources with respect to the previous (consistent) version. Unfortunately, as pointed out by the author too, the approach is not based on a theoretical foundation, and therefore there is no proof of soundness. So, it might happen that *Javamake* fails to force the recompilation of some classes which is actually needed for ensuring the consistency of the project. Furthermore, *Javamake* cannot avoid a considerable amount of unnecessary recompilations. The main advantages of *Javamake* are that it is well documented and in practice its implementation can work upon any Java compiler.

Our system can be exploited for extending *Javamake* in order to sensibly decrease the number of unnecessary recompilations: a set of local type assumptions for a source fragment *C* can be automatically inferred when compiling *C* starting from a standard environment. These local type assumptions describe in a precise way what *C* needs in order to be recompiled into the same binary. These assumptions can be used to decide whether an unchanged class needs to be recompiled. In fact, if a new global environment still entails the local assumptions for a fragment *S*, then there is no need to recompile *S*. From the complexity point of view, checking whether a local assumption is entailed by a standard environment requires a polynomial algorithm that can be implemented efficiently [2]. Without going into the details, let us show the idea on the the example already discussed in Section 2:

```

class P extends Object {
    int f(Object o) { return 0 ; }
    // int f(int i) { return i ; }
}
class U extends Object {}
class Y extends X {}
class X extends Object {}

class H extends P {
    int g(P p) {return p.f(new(H));}
    int m() {return new H().g(new P());}
    U id(U u){return u;}
    X em(Y y){return y;}
}

```

These classes compile successfully, and they form our example project. The compiler would generate the following type assumption for *H*:

$$\Gamma_l = \exists P, \exists U, \exists X, \exists Y, P \not\leq H, P \odot \text{int } g(P), P \odot \text{int } m(), \\ P \odot U \text{ id}(U), P \odot X \text{ em}(Y), P.f(H) \xrightarrow{\text{res}} \langle \text{Object}, \text{int} \rangle, Y \leq X$$

If we add a method `f(int)` in `P` (that is, if we remove the comment in the previous listing), then class `H` would still call the same method as before, because the new method is not even applicable to the call with an argument of type `P`. These considerations are formally captured by the entailment relation: the new standard environment (that can be extracted from the new source for `P` and the old binaries of the other classes) still entails Γ_l therefore there is no need to recompile `H`. For instance, the reader can verify that $P.f(H) \xrightarrow{\text{res}} \langle \text{Object}, \text{int} \rangle$ can still be entailed. On the other hand, whereas *Javamake*⁵ is able to detect that classes `U`, `X` and `Y` need not to be recompiled, since they do not use `P` at all, it cannot distinguish between changes to a set of overloaded methods that alter the resolution of a particular call and changes that do not. So, *Javamake* would unnecessarily recompile class `H`, because it contains a call to `P.f`, producing the same binary as before.

4 Conclusion and Other Related Work

We have presented a new system for Java-like languages for better supporting separate compilation; the system relies on the notions of local type environment and entailment of type environments. In comparison with the standard type environments adopted by SDK systems, local type environments allow specification of weaker type assumptions for compiling in isolation a code fragment.

The system has been proved to be compositional and strictly stronger (and, therefore, more accurate) than SDK systems [2], and to have principal typings [5]. Furthermore, local type assumptions can be automatically generated the first time a program is compiled and can be used later to avoid unnecessary recompilations; based on these last considerations, in Section 3 we have sketched an algorithm for improving the Java-specific make technology *Javamake* [8].

Separate compilation is an issue that has been deeply studied for programming languages, especially in the context of *selective recompilation* [1] which seeks to reduce rebuilding time due to source modifications.

Besides the paper on *Javamake* [8] already discussed in Section 3, there is a number of papers on selective recompilation for several languages. According to the classification given in [1], [11] adopts for ML an approach which involves both cut-off elimination and smart recompilation, while [16] investigates smartest recompilation, by employing type inference to derive the type assumptions needed for compiling an ML code fragment in isolation. Smart and smarter recompilations have been considered as well for C-like languages [17, 15].

Unfortunately, very little has been done on this side for Java-like languages. This paper, together with [6, 3, 4, 8, 13], is a step towards a better understanding of separate compilation of Java-like languages.

The solution presented here is similar to attribute recompilation, according to the classification given in [1]. Here attributes correspond to local type assumptions which can be automatically inferred when compiling a closed set of fragments; these assumptions can be used later for selective recompilation.

⁵ Version 1.3.1, the latest available at the moment of writing this paper.

Acknowledgements We warmly thank Sophia Drossopoulou and Elena Zucca for their useful suggestions and corrections to previous versions of this paper.

References

1. Rolf Adams, Walter Tichy, and Annette Weinert. The cost of selective recompilation and environment processing. *ACM Transactions on Software Engineering and Methodology*, 3(1):3–28, January 1994.
2. D. Ancona and G. Lagorio. Stronger typings for separate compilation of Java-like languages. Technical report, DISI, March 2003.
3. D. Ancona, G. Lagorio, and E. Zucca. A formal framework for Java separate compilation. In B. Magnusson, editor, *ECOOP 2002 - Object-Oriented Programming*, number 2374 in Lecture Notes in Computer Science, pages 609–635. Springer, 2002.
4. D. Ancona, G. Lagorio, and E. Zucca. True separate compilation of Java classes. In *ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP’02)*, pages 189–200. ACM Press, 2002.
5. D. Ancona and E. Zucca. Principal typings for Java-like languages. To appear as DISI technical report.
6. D. Ancona and E. Zucca. True modules for Java-like languages. In J.L. Knudsen, editor, *ECOOP’01 - European Conference on Object-Oriented Programming*, number 2072 in Lecture Notes in Computer Science, pages 354–380. Springer, 2001.
7. L. Cardelli. Program fragments, linking, and modularization. In *ACM Symp. on Principles of Programming Languages 1997*, pages 266–277. ACM Press, 1997.
8. M. Dmitriev. Language-specific make technology for the Java programming language. *ACM SIGPLAN Notices*, 37(11):373–385, 2002.
9. S. Drossopoulou and S. Eisenbach. Describing the semantics of Java and proving type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, number 1523 in Lecture Notes in Computer Science, pages 41–82. Springer, 1999.
10. S. Drossopoulou, G. Lagorio, and S. Eisenbach. Flexible models for dynamic linking. In *European Symposium on Programming 2003*, 2003.
11. R. Harper, P. Lee, F. Pfenning, and E. Rollins. A compilation manager for standard ML of New Jersey. In *ACM SIGPLAN Workshop on Standard ML and its Applications*, July 94.
12. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1999*, pages 132–146, November 1999.
13. G. Lagorio. Towards a smart compilation manager for Java. In *Italian Conf. on Theoretical Computer Science 2003*, Lecture Notes in Computer Science. Springer, 2003. To appear.
14. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Second edition, 1999.
15. Robert W. Schwanke and Gail E. Kaiser. Smarter recompilation. *ACM Transactions on Programming Languages and Systems*, 10(4):627–632, October 1988.
16. Z. Shao and A.W. Appel. Smartest recompilation. In *ACM Symp. on Principles of Programming Languages 1993*, pages 439–450. ACM Press, 1993.
17. Walter F. Tichy. Smart recompilation. *ACM Transactions on Programming Languages and Systems*, 8(3):273–291, July 1986.
18. J.B. Wells. The essence of principal typings. In *Proc. 29th Int’l Coll. Automata, Languages, and Programming (ICALP’02)*, number 2380 in Lecture Notes in Computer Science, pages 913–925. Springer, 2002.

Static analysis for eager stack inspection [★]

Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari

Dipartimento di Informatica, Università di Pisa, Italy
{bartolet, degano, giangi}@di.unipi.it

Abstract. The paper focuses on stack inspection, a mechanism for the implementation of security-aware programming languages. We introduce a static analysis which safely approximates the set of access rights granted to code at run-time. This analysis allows for fast implementations of stack inspection which based on an (hyper-) eager strategy.

1 Introduction

In the Java security model, access control decisions are taken by examining the call stack at run-time. A permission is granted, provided that it belongs to *all* principals on the call stack. The so-called *privileged operations* are an exception. These are allowed to execute any code granted to their principal, regardless of the calling sequence. This access control mechanism is known as *stack inspection*.

There are at least two strategies for implementing stack inspection:

- in the *eager evaluation* strategy, the set of granted permissions is updated at each method call (and return).
- in the *lazy evaluation* strategy, the call stack is retrieved and inspected only when an access control is performed.

In the eager strategy, security checks can be performed very efficiently; on the other hand, there is an overhead at each cross-domain method invocation. Since security checks are statistically less frequent than cross-domain calls, actual implementations of the Java virtual machine adopt the lazy strategy. However, this choice could be expensive. First, the run-time overhead due to the analysis of stack frames may grow very high. Second, lazy stack inspection deeply affects standard program optimizations, such as method inlining and (general) tail call elimination.

We consider here an idealized object-oriented language with primitive constructs for method invocations, exceptions, and access control. We represent programs by *control flow graphs*, an abstract programming model not tied to any particular programming language.

We define a control flow analysis [11] over these graphs, called *Trace Permissions Analysis*. Intuitively, the analysis computes, for any program point and any execution reaching that point, the set of permissions granted at run-time.

[★] Work partially supported by EU project DEGAS (IST-2001-32072), MIUR project “MEtodi FormalI per la Sicurezza e il TempO” (MEFISTO), and MIUR project “Network Aware Programming: Object, Languages, Implementation” (NAPOLI).

The Trace Permissions Analysis allows for effective implementations of eager stack inspection, on which this paper concentrates. Also, the analysis provides us with the basis for several code optimizations, such as method inlining, tail call elimination, removal of redundant security checks, and dead-code elimination.

Because of space limitations, here we will provide the overall picture of the technical development of our approach by focusing on the underlying ideas. We refer to the full paper [2] for a detailed description of the model, the proofs of our statements, some illustrative examples, and for code optimizations for the lazy strategy.

2 The program model

We model Java-like programs as *control flow graphs* (CFGs for short) whose nodes represent the activities relevant for stack inspection (i.e. checks, method invocations and returns) and whose arcs represent the flow of control. We do not define how CFGs are extracted from an actual program. This construction is well understood and algorithms and tools exist for it; see e.g. [9, 11, 15, 16].

By construction, CFGs hide any data flow information, and are therefore approximated; typically, switching is rendered as non-deterministic choice. This approximation is *safe*, in the sense that any actual execution flow is represented by a path in the CFG. However, the converse may not be true: some paths may exist which do not correspond to any actual execution. For instance, both branches of an “if” statement are represented, even if always the same branch is taken at run-time.

There is a further source of approximation, especially for object-oriented languages with dynamic resolution of method invocations. In Java, for example, when a program invokes an instance method on an object O , the virtual machine may have to choose among various implementations of that method. The decision is not based on the declared type of O , but on the actual class O belongs to, which is unpredictable at static time. To be safe, CFGs consider a superset of the methods that can be invoked at each program point. This is a main source of approximation for the analyses built over CFGs.

2.1 Syntax

Let $\mathcal{D}$ be a finite set of protection domains, and $\mathcal{P}$ be a finite set of permissions. CFGs are defined as follows.

Definition 1. A CFG $\langle N \cup \{n_\varepsilon\}, E, Priv, Dom \rangle$ is an oriented graph, where:

- N is the set of nodes. Each node $n \in N$ is associated with a label $\ell(n)$, describing the control flow primitive it represents. Labels partition nodes in three kinds: *call nodes*, that stand for method invocation, *return nodes*, which represent return from a method, and *check nodes*, which enforce the access control policy. For each $P \in \mathcal{P}$, a node labeled $check(P)$ can be seen as the abstract representation of an `AccessController.checkPermission(P)`

instruction in the Java language. The distinguished element $n_\varepsilon \notin N$ plays the technical role of a single, isolated entry point.

- $E \subseteq (N \cup \{n_\varepsilon\}) \times N$ is the set of edges. Edges are partitioned into four sets: entry edges $\bullet \rightarrow n$, that represent the entry points of a program; call edges $n \rightarrow n'$, which model interprocedural flow; transfer edges $n \dashrightarrow n'$, which correspond to sequencing; and catch edges $n \dashrightarrow_\dagger n'$, which correspond to exception handling. The last two kinds of edges represent intraprocedural flow. The set of entry edges contains all pairs (n_ε, n) where n is a program entry point. The n_ε element is the source of entry edges, only.
- $\text{Priv} : N \rightarrow \text{Bool}$ tells whether a node enables its privileges or not.
- $\text{Dom} : N \rightarrow \mathcal{D}$ is a mapping from nodes to protection domains.

When unambiguous, we shall write $\langle N, E \rangle$ instead of $\langle N \cup \{n_\varepsilon\}, E, \text{Priv}, \text{Dom} \rangle$.

Each CFG is associated with a *security policy* $\text{Perm} : \mathcal{D} \rightarrow 2^{\mathcal{P}}$, which grants a set of permissions to each protection domain. Hereafter, we will always abbreviate $\text{Perm}(\text{Dom}(n))$ with $\text{Perm}(n)$.

Definition 2. The methods of a CFG $\langle N, E \rangle$ are the connected components of the graph $\langle N, E' \rangle$, where E' is the set of intraprocedural edges in E , with no orientation. We call $\mu(n)$ the method to which node n belongs. The entry points of $\mu(n)$ are defined as:

$$\varepsilon(\mu(n)) = \{n' \in \mu(n) \mid \bullet \rightarrow n' \vee \exists m \in N. m \rightarrow n'\}$$

The set $\rho(n)$ of return nodes associated to a node n is:

$$\rho(n) = \{m \in N \mid \ell(m) = \text{return} \wedge n \rightarrow \varepsilon(\mu(m))\}$$

The set $\xi(n)$ of nodes that may throw an exception catchable by n is defined as the smallest set satisfying:

$$\xi(n) = \begin{cases} \{n\} & \text{if } \ell(n) = \text{check}(P) \\ \{\xi(n') \mid n \rightarrow \varepsilon(\mu(n')) \wedge n' \not\rightarrow_\dagger\} & \text{otherwise} \end{cases}$$

The set $\xi_1(n)$ of nodes that may propagate an exception to n is defined as:

$$\xi_1(n) = \{n' \mid n \rightarrow \varepsilon(\mu(n')) \wedge n' \not\rightarrow_\dagger \wedge \xi(n') \neq \emptyset\}$$

As discussed in [2], all the CFGs derived from admissible Java programs satisfy the following well-formedness constraints: (1) check nodes have no outgoing call edges; (2) return nodes have no outgoing edges; (3) each method has a single entry point (4) nodes in the same method are in the same protection domain. Moreover, we require that only call nodes can be privileged. In general, security checks can also occur within privileged actions: however, privileged check nodes make little sense, because it is always possible to determine whether a privileged check will succeed or not. Similarly, there is no point in enabling return nodes to be privileged, because a return node will never be on the call stack when stack inspection is performed.

$\frac{\bullet \rightarrow n}{[] \triangleright [n]}$	$\frac{\ell(n) = \text{call} \quad n \rightarrow n'}{\sigma : n \triangleright \sigma : n : n'}$	$\frac{\ell(m) = \text{return} \quad n \dashrightarrow n'}{\sigma : n : m \triangleright \sigma : n'}$
$\frac{\ell(n) = \text{check}(P) \quad \sigma : n \vdash P \quad n \dashrightarrow n'}{\sigma : n \triangleright \sigma : n'}$	$\frac{\ell(n) = \text{check}(P) \quad \sigma : n \not\vdash P}{\sigma : n \triangleright \sigma : n \not\vdash}$	
$\frac{n \dashrightarrow_i n'}{\sigma : n \not\vdash \triangleright \sigma : n'}$	$\frac{n \not\vdash_i}{\sigma : n \not\vdash \triangleright \sigma \not\vdash}$	
$\frac{}{[] \vdash P}$	$\frac{P \in \text{Perm}(n) \quad \sigma \vdash P}{\sigma : n \vdash P}$	$\frac{P \in \text{Perm}(n) \quad \text{Priv}(n)}{\sigma : n \vdash P}$

Table 1. Operational semantics of CFGs.

2.2 Semantics

The operational semantics of CFGs is defined by a transition system whose configurations are sequences of nodes, modeling call stacks. Additionally, each state has a boolean tag which tells whether an exception is *active*, i.e. thrown and not caught yet. Formally, we define the set of states as $N^* \times \text{Bool}$.

If no exception is active, a state is represented as sequence of nodes enclosed in square brackets: for example, $\sigma = [n_0, \dots, n_k]$ is a state whose top node is n_k . If an exception is active, we append the symbol $\not\vdash$ to the sequence of nodes, i.e. $\sigma \not\vdash$ abbreviates $\langle \sigma, \text{true} \rangle$. Pushing a node n on a stack σ is written as $\sigma : n$ (the infix operator $:$ associates to the left).

The transition relation $\triangleright$ between states is the minimal relation induced by the inference rules in Table 1. A *trace* of G leading to $\langle \sigma_k, x_k \rangle$ is a derivation $\langle \sigma_0, x_0 \rangle \triangleright \dots \triangleright \langle \sigma_k, x_k \rangle$ where $\sigma_0 = []$ and $x_0 = \text{false}$. By overloading the notation, we also denote with $\triangleright$ the relation:

$$\frac{}{G \triangleright \langle [], \text{false} \rangle} \qquad \frac{G \triangleright \langle \sigma, x \rangle \quad \langle \sigma, x \rangle \triangleright \langle \sigma', x' \rangle}{G \triangleright \langle \sigma', x' \rangle}$$

stating when there is a trace of G which can lead to a given state. We say that a node n is reachable iff $\langle \sigma : n, x \rangle$ is a reachable configuration.

In our formalization, we use a slightly simplified version of the full access control algorithm presented in [8]. The simplified algorithm scans the call stack top-down. Each frame in the stack refers to the protection domain containing the class to which the called method belongs. As soon as a frame is found whose protection domain has not the required permission, an `AccessControlException` is

$TP_{in}(n) = \bigcup_{(m,n) \in E} TP_{out}(m, n)$	
$TP_{out}(m, n) = \begin{cases} \{\{\text{Dom}(n)\}\} & \text{if } \bullet \rightarrow n \\ \{\gamma \cup \{\text{Dom}(n)\} \mid \gamma \in TP_{call}(m)\} & \text{if } m \rightarrow n \\ TP_{trans}(m) & \text{if } m \dashrightarrow n \\ TP_{catch}(m) & \text{if } m \dashrightarrow_i n \end{cases}$	
$TP_{call}(n) = \begin{cases} \{\{\text{Dom}(n)\}\} & \text{if } \text{Priv}(n) \text{ and } TP_{in}(n) \neq \emptyset \\ TP_{in}(n) & \text{otherwise} \end{cases}$	
$TP_{trans}(n) = \begin{cases} \{\gamma \in TP_{in}(n) \mid P \in \Pi(\gamma)\} & \text{if } \ell(n) = \text{check}(P) \\ \{\gamma \in TP_{in}(n) \mid \text{Trans}(n, \{\text{Dom}(n)\})\} & \text{if } \ell(n) = \text{call}, \text{Priv}(n) \\ \{\gamma \in TP_{in}(n) \mid \text{Trans}(n, \gamma)\} & \text{otherwise} \end{cases}$	
$TP_{catch}(n) = \begin{cases} \{\gamma \in TP_{in}(n) \mid P \notin \Pi(\gamma)\} & \text{if } \ell(n) = \text{check}(P) \\ \{\gamma \in TP_{in}(n) \mid \text{Catch}(n, \{\text{Dom}(n)\})\} & \text{if } \ell(n) = \text{call}, \text{Priv}(n) \\ \{\gamma \in TP_{in}(n) \mid \text{Catch}(n, \gamma)\} & \text{otherwise} \end{cases}$	
<hr/> $\begin{aligned} \text{Trans}(n, \gamma) &\stackrel{def}{=} \exists m \in \rho(n). \gamma \cup \{\text{Dom}(m)\} \in TP_{in}(m) \\ \text{Catch}(n, \gamma) &\stackrel{def}{=} \exists m \in \xi_1(n). \gamma \cup \{\text{Dom}(m)\} \in TP_{catch}(m) \end{aligned}$	

Table 2. Flow equations for the TP analysis.

raised. The algorithm succeeds when a privileged frame is found that carries the required permission, or when all frames have been visited. We formally specify this behavior by the minimal relation induced by the inference rules for $\vdash$ in Table 1. We say that a permission P is *granted* to a state σ if $\sigma \vdash P$.

3 The Trace Permissions Analysis

We introduce the Trace Permissions Analysis (TP), a static analysis over CFGs which approximates the access rights granted to each reachable state.

Since the set of permissions granted to a state is just the intersection of the permissions associated to each protection domain traversed after the last privileged frame (if any), we can identify the set $\{P \in \mathcal{P} \mid \sigma \vdash P\}$ with the

security context $\Gamma(\sigma)$, where $\Gamma : N^* \rightarrow 2^{\mathcal{D}}$ is defined as follows:

$$\Gamma([]) = \emptyset \quad \Gamma(\sigma : n) = \begin{cases} \{\text{Dom}(n)\} & \text{if } \text{Priv}(n) \\ \Gamma(\sigma) \cup \{\text{Dom}(n)\} & \text{otherwise} \end{cases}$$

The set of permissions granted to a security context γ is:

$$\Pi(\gamma) = \bigcap_{D \in \gamma} \text{Perm}(D)$$

The permissions granted to the security context of a state σ are exactly the permissions granted to σ , i.e. $\sigma \vdash P \iff P \in \Pi(\Gamma(\sigma))$ for all $\sigma \in N^*$, $P \in \mathcal{P}$.

Given a CFG G and a security policy Perm , the analysis is specified by the set of equations $TP^=(G, \text{Perm})$ in Table 2. A solution $\tau \models TP^=(G, \text{Perm})$ is a 5-tuple $\tau = \langle \tau_{in}, \tau_{call}, \tau_{trans}, \tau_{catch}, \tau_{out} \rangle$ which satisfies all the equations. The purpose of the analysis is to find, for each node n , the set $\{\Gamma(\sigma : n) \mid G \triangleright \sigma : n\}$.

Technically, TP is a forward, monotone data flow analysis with values in $2^{2^{\mathcal{D}}}$. Since both G and $\mathcal{D}$ are finite, the least solution to the analysis does exist and is finitely computable.

The following theorem states the correctness of the TP analysis. The first equation below states that any solution to the analysis is sound w.r.t. the operational semantics. The second equation states that the least solution to the analysis is also complete. This fact should not seem bizarre: indeed, completeness is only up to the precision of the CFG, which is an approximated model of the analyzed program.

Theorem 1. *Let $\tau \models TP^=(G, \text{Perm})$. Then:*

$$G \triangleright \sigma : n \implies \exists \gamma \in \tau_{call}(n). \gamma = \Gamma(\sigma : n)$$

Moreover, the minimal solution w.r.t. the inclusion relation on $2^{2^{\mathcal{D}}}$ is such that:

$$\gamma \in \tau_{call}(n) \implies \exists \sigma. G \triangleright \sigma : n \wedge \gamma = \Gamma(\sigma : n)$$

The worklist algorithm which actually computes the (unique) minimal solution to the analysis has computational complexity $\mathcal{O}(c \cdot |N|) = \mathcal{O}(|N|)$. The constant c depends on the number of protection domains occurring in G : in the worst case, $c = 2^{3 \cdot |\mathcal{D}_G|}$, where $\mathcal{D}_G = \bigcup_{n \in N} \text{Dom}(n)$. However, the exponential factor only occurs when the number of protection domains is proportional to the number of nodes. Actually, the number of protection domains can be considered as a constant, because it depends on the security policy, rather than on the size of the program.

Dynamic linking is the mechanism which allows a program to be extended on demand, e.g. with code coming from the network. Although our program model does not directly support this feature, the TP analysis can be computed incrementally. The CFG construction algorithm is demanded to correctly perform the

dynamic linking of the relevant CFGs, as in [14]. Indeed, this operation cannot be performed by looking at the CFGs alone, because CFGs do not carry enough information to restrict the set of targets of dynamically dispatched method invocations.

We briefly show how the incremental computation of the analysis is performed. Let $G = \langle N, E \rangle$, and assume that a solution τ to $TP^=(G, Perm)$ is available when the CFG $G' = \langle N', E' \rangle$ is loaded. Through the CFG construction algorithm, we single out the set $E_{\bowtie}$ of *resolved calls* between G and G' , i.e. those edges $n \rightarrow m$ such that n, m do not belong both to the same CFG. Linking G and G' together yields the CFG $G \bowtie G' = \langle N \cup N', E \cup E' \cup E_{\bowtie} \rangle$. The analysis $\tau' \models TP^=(G \bowtie G', Perm)$ is a refinement of τ . To compute it, the worklist algorithm adds to τ all the contexts associated with the new paths created by the resolved calls. It suffices now to restart the algorithm with the worklist containing all nodes n such that, for some node m , $(n, m) \in E_{\bowtie}$. Moreover, the worklist must include all entry points of G' , if any. Although our approach is not fully compositional, note that adding new executable paths to a CFG never affects the analysis of the old ones.

4 Eager stack inspection

We now specify an alternative implementation of eager stack inspection, which exploits our TP analysis to efficiently update the security contexts.

We adopt the *security passing style* of [18] to track the security context as an additional parameter of each method invocation. The type of this parameter is assumed to be one of the primitive integral types of the JVM (i.e. `byte`, `short`, `int` and `long`); accordingly, its size (in bits) is then $k \in \{8, 16, 32, 64\}$.

Choose a set $\mathcal{D}_0 = \{D_1, \dots, D_{k-1}\} \subseteq \mathcal{D}$ of protection domains. If $\mathcal{D}$ has not enough elements, add the needed ones, and assign them arbitrary permissions.

Given a CFG G and a security policy $Perm$, a solution to $TP^=(G, Perm)$ can be used to enumerate the set $\{I(\sigma) \mid G \triangleright \sigma\}$ of the reachable security contexts. Let $\gamma_0, \dots, \gamma_p$ be such an enumeration. Represent now a security context γ_i as a k -bits array $\alpha_i = \langle \alpha_{i,0}, \dots, \alpha_{i,k-1} \rangle$, where:

- if $\gamma_i \subseteq \mathcal{D}_0$, then $\alpha_{i,0} = 0$ and, for each $j \in 1..k-1$, $\alpha_{i,j} = 1$ iff $D_j \in \gamma_i$.
- otherwise, $\alpha_{i,0} = 1$ and $\langle \alpha_{i,1}, \dots, \alpha_{i,k-1} \rangle$ is the binary representation of i .

Security contexts are updated at each method invocation and return. The intuition is that $2^{\mathcal{D}_0}$ contains the contexts that we expect to occur with high probability in executions, and therefore require very efficient updating. These contexts are represented as arrays of bits: the i -th bit is set iff the protection domain D_i has been traversed.

The contexts outside $2^{\mathcal{D}_0}$ are represented by their indexes in the enumeration computed by the TP analysis. The transition function h between security contexts is cached in a hash table; h is computed as a side effect when constructing a solution to the TP analysis. Formally, $h(i, n, n') = j$ whenever γ_j is the context of the state obtained when the control flows from n to n' , starting from a state

with context γ_i . There is no need to store the entries of h where both γ_i and γ_j in $2^{\mathcal{D}_0}$.

Formally, the context updating operations are implemented as follows:

- on method invocation, let n be a call to $\mu(n')$ and the current security context be γ_i . If $\gamma_i \subseteq \mathcal{D}_0$ and $\text{Dom}(n') = D_j$, then the new context is $\langle 0, \alpha_{i,1}, \dots, \alpha_{i,k-1} \rangle \vee 2^{k-j-1}$ (bitwise or). That is, $\alpha_{i,j}$ is set to indicate that D_j has been traversed. Otherwise, the new context is $\alpha_{h(i,n,n')}$.
- on method return, the context is retrieved from the popped call stack.

Security checks are performed by looking at the current context, instead of inspecting the call stack. The intuition is that, for the contexts in $2^{\mathcal{D}_0}$, a check for permission P_j succeeds iff no protection domain D_i with $P_j \notin \text{Perm}(D_i)$ has been traversed.

More formally, let $P_1, \dots, P_q$ be the set of permissions checked in G . For each $j \in 1..q$, we define a k -bits array $\beta_j = \langle 0, \beta_{j,1}, \dots, \beta_{j,k-1} \rangle$ as follows:

$$\beta_{j,i} = \begin{cases} 0 & \text{if } P_j \in \text{Perm}(D_i) \\ 1 & \text{otherwise} \end{cases}$$

Let n be a check for permission P_j , γ_i be the current context, and let n' follow n sequentially. If $\gamma_i \subseteq \mathcal{D}_0$, then the check succeeds iff $\alpha_i \wedge \beta_j = \mathbf{0}$ (bitwise and). Otherwise, the check succeeds iff $h(i, n, n')$ is defined.

Compared with the lazy evaluation strategy, our technique involves an overhead at each method invocation: besides the cost of passing an additional parameter, we have to perform either a “bitwise or” operation (or a hash table lookup). Security checks require a “bitwise and” (or a hash table lookup), and are independent of the size of the call stack. As a matter of fact, a good choice of the set $\mathcal{D}_0$ is crucial, and possibly requires a statistical estimate of the frequency of contexts. The problem of statically determining the probability of execution traces in CFGs seems to be unsolvable in the general case. However, we hope that a rough approximation of the most frequent states will suffice. Some form of dynamic analysis can possibly be carried on to refine that approximation.

5 Conclusions and related work

We have developed a static analysis for the Java bytecode. The Trace Permissions Analysis computes a safe approximation to the set of permissions which are *always* granted to bytecode at run-time. The analysis is sound and complete w.r.t. the CFGs derived from the bytecode (however, these graphs only approximate the actual behavior). Our analysis makes various optimizations possible; here, we only focused on eager stack inspection. We restricted our attention to Java, but the same techniques can be applied to other programming languages whose authorization mechanisms rely on stack inspection (e.g. C# [19]).

Many authors advocated the use of static techniques in order to understand and optimize stack inspection.

Besson, Jensen, Le Métayer and Thorn [4] formalize classes of security properties through a linear-time temporal logic. They show that a large class of policies (including stack inspection) can be expressed in this formalism. Model checking is then used to prove that local security checks enforce a given global security policy. The verification method can be used to optimize stack inspection by eliminating the redundant checks. This is done by computing, for each node n , the set $\{P \in \mathcal{P}_{check} \mid G \triangleright \sigma : n \wedge \sigma : n \vdash P\}$, where $\mathcal{P}_{check}$ is the set of permissions checked in G . The computational complexity of the method is $\mathcal{O}(c \cdot |N|)$, where the constant c depends on the cardinality of $\mathcal{P}_{check}$ (in the worst case, $c = 2^{|\mathcal{P}_{check}|}$). Therefore, our TP analysis performs better when there are few protection domains, while [4] is more efficient when there are few security checks. Note that our analysis is at least as precise as [4], because $\mathcal{P}_{check} \subseteq \mathcal{P}$. Also, the analysis in [4] does not seem to scale up smoothly to handle dynamic linking, because it must be recomputed each time a new permission is discovered.

Wallach, Appel and Felten [18] formalize stack inspection by exploiting the access control logic of [1]. The authors show that their decision procedure is equivalent to Java stack inspection, according to an informal operational semantics. Moreover, they propose an alternative semantics of eager stack inspection, called *security-passing style*. This technique consists of tracking the security state of an execution as an additional parameter of each method invocation. This allows for interprocedural compiler optimizations that do not interfere with stack inspection. The security-passing style allows each security operation to be performed in constant time, but it involves an overhead, because the security state must be computed at each method invocation. Dynamic caching techniques are adopted to reduce this overhead: therefore, in the optimal case, the additional cost of each method invocation is that of a hash lookup. The same technique allows for an implementation of security checks which requires a hash lookup in the optimal case. Instead, in our approach, each security operation costs as a hash lookup *in the worst case*, while, in the optimal case, it costs as a cheap bitwise operation. A further difference w.r.t our approach is that [18] assumes that the whole program is available at compilation time.

Pottier, Skalka and Smith [12] address the problem of stack inspection in λ_{sec} , a typed lambda calculus enriched with primitive constructs for enforcing security checks and managing permissions. Stack inspection never fails on a well typed program, because the set of permissions granted at runtime always includes the security context. These types are very powerful and can deal with several issues (e.g. security policy overriding, dependencies from untrusted code). Moreover, they can be smoothly extended to deal with objects by standard type-theoretic techniques. The problem of establishing the correctness of program transformations in presence of stack inspection is investigated by Fournet and Gordon in [7]. They present an equational theory, together with a coinductive proof technique, for a λ -calculus enriched with access control primitives, as in [12]. They study how stack inspection affects program behavior, proving that certain function in-linings and tail-call eliminations are correct. The theory is used to reason about the (somewhat limited) security properties guaranteed by stack inspection.

References

1. M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 4(15):706–734, Sept. 1993.
2. M. Bartoletti, P. Degano, and G. Ferrari. Stack inspection and program transformations. <http://www.di.unipi.it/~bartolet/static-stack.ps>, 2003.
3. L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. In *Foundations of Computer Security (FCS '02)*, Copenhagen, July 2002.
4. F. Besson, T. Jensen, D. Le Métayer, and T. Thorn. Model checking security properties of control flow graphs. *Journal of computer security*, 9:217–250, 2001.
5. J. Clemens and M. Felleisen. A tail-recursive semantics for stack inspections. In P. Degano, editor, *Proc. ESOP'03*, volume 2618 of *LNCS*. Springer-Verlag, 2003.
6. U. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, 2000.
7. C. Fournet and A. D. Gordon. Stack inspection: theory and variants. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 307–318. ACM Press, 2002.
8. L. Gong. *Inside Java 2 platform security: architecture, API design, and implementation*. Addison-Wesley, 1999.
9. D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM TOPLAS*, 23(6), Nov. 2001.
10. Microsoft Corp. *.NET Framework Developer's Guide: Securing Applications*.
11. F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.
12. F. Pottier, C. Skalka, and S. Smith. A systematic approach to static access control. In D. Sands, editor, *Proceedings of the 10th European Symposium on Programming (ESOP'01)*, volume 2028 of *LNCS*, pages 30–45. Springer-Verlag, Apr. 2001.
13. F. B. Schneider. Enforceable security policies. Technical Report TR98-1664, Cornell University, Jan. 1998.
14. A. Souter and L. Pollack. Incremental call graph reanalysis for object-oriented software maintenance. In *IEEE International Conference on Software Maintenance*, Nov. 2001.
15. V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000)*, volume 35(10) of *SIGPLAN Notices*, pages 264–280, Minneapolis, MN, Oct. 2000. ACM Press.
16. F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '00)*, pages 281–293, 2000.
17. D. Walker. A type system for expressive security policies. In *Conference record of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 254–267. ACM Press, 2000.
18. D. S. Wallach, A. W. Appel, and E. W. Felten. SAFKASI: a security mechanism for language-based systems. *ACM TOSEM*, 9(4):341–378, Oct. 2001.
19. C. Wille. *Presenting C#*. SAMS Publishing, 2000.

Abstraction-carrying Code: a New Method to Certify Temporal Properties

Songtao Xia and James Hook

OGI School of Science & Engineering
Oregon Health & Science University
Beaverton OR 97006, USA

Abstract. Temporal properties represent an important class of properties that, when verified, help to establish high confidence of a program. Certification of temporal properties provides such confidence to a client of a mobile program. Current techniques adopt a proof of a temporal property as a certificate; they therefore suffer from the state-explosion problem. Using these techniques, we may not afford the certification of multiple properties.

We propose abstraction-carrying code (ACC). This approach adopts an abstract interpretation of a program as a certificate. A client will validate and model check the abstraction to verify the temporal properties.

This paper provides an overview of ACC. We summarize our early results, which include 1) how we generate an abstraction for an intermediate program, 2) how we represent an abstraction in an easily verifiable way, and 3) experience of application of ACC to a few industrial strength problems. The paper also discusses new perspectives made possible by the ACC framework.

1 Introduction

In the motivating scenario the developer of a mobile application develops software anticipating remote verification. The software is packaged with checkable annotations. These annotations are communicated with the software to the client, who uses the annotations to validate required temporal properties. These properties range from data-flow safety properties, such as the correct use of APIs, to liveness properties.

Research on proof-carrying code, initiated by Necula and Lee, addresses the problem of how a client trusts an untrusted program by verifying a mathematical proof for certain safety properties [17]. Early variants of PCC, including TAL [16, 15], DTAL [21], ECC [13], Certified Binary [19], FPCC [1], TouchStone and others, focus primarily on type safety and memory safety.

Researchers have begun to explore extending PCC to certify temporal properties. For example, Temporal PCC (TPCC) is able to certify and verify temporal properties [3]. It represents the traditional line of thinking inherited from PCC, which is catch-phrased by “proof as certificate”. The success of PCC may have

allured people to work on proofs when certificates are needed. For example, systems are proposed, and in some case implemented, to construct a proof from the result of a model checker [11, 12, 2, 20].

These proof-based certification methods all suffer from the state explosion problem. The size of a proof tends to expand drastically as the problem size grows. Because of this problem, we cannot cash in a lot of applications made possible by the expressive power of temporal properties. Examples include fully automated certification of general temporal properties, including liveness properties, of concurrent programs. Because of the same problem, with limited network bandwidth, a server may not be able to certify multiple properties at one time, which is one of most demanding requirement in a realistic system. Furthermore, in a proof based certification system, the certificate is tightly coupled with the property of concern: a proof for one property proves that property only. This fact makes it impossible to keep a server from knowing the exact security policy running by a client.

The approach presented here, partially inspired by SLAM, Bandera and other related projects, adopts predicate abstraction as the certification method. More specifically, on the server side, we adopt Boolean abstraction [8, 5, 2] to compute an abstract model of a source program. Then we compile the source program, in an abstraction-preserving fashion, to intermediate program. The result of the compilation is an intermediate program that has the same Boolean abstraction as the source program. The server will send the boolean programs, encoded as index type annotations, as the certificate to the client. The client will use an index type checker to validate the abstraction before it model checks the temporal properties on the abstract model. Index types are a version of refinement types that augment conventional type checking with a constraint satisfaction mechanism providing some of the expressive power of dependent types. In one index type system, not only can we verify the abstract model but we also verify system invariants. We have implemented a prototype infrastructure of ACC in a toolkit called ACCEPT (ACC Evaluation Prototype Toolkit). An illustration of the infrastructure is in Fig.1.

Fig. 1. Relation between source program, assembly program, boolean program and index type

This paper provides an overview of ACC. We summarize our early results, which include 1) how we generate an abstraction for an intermediate program,

2) how we represent an abstraction in an easily verifiable way, and 3) experience of application of ACC to a few industrial strength problems. The paper also discusses new perspectives made possible by the ACC framework.

The paper is structured as follows. In Section 2, we introduce a few examples that motivate our investigation. Next we present the framework of ACC in terms of control flow automata (CFA). In Section 4 we summarize our experience with applying two versions of ACC implementation to two categories of programs, including the certification of industrial strength programs. Certification of multiple properties is not yet implemented; we discuss related technical issues in Section 5. Finally we conclude.

2 Motivation Examples

2.1 Example 1: Concurrency

On the left side of Fig. 2 we list the code fragment of a bounded buffer program written in a Java-like language. There are two symmetric threads, the figure showing one of them. Global variables `buf1` and `buf2` count the elements in the buffers. The capacity of both buffers is a constant `bound`. The thread shown constantly moves elements from one buffer to the other. The other thread does the opposite. One of the properties to expect is that whenever a buffer is full, then sometime in the future one of the elements must be removed. This property, which we name as `FullToNonFull`, can be expressed in a Linear Temporal Logic (LTL) term as

$$\Box(\text{Full} \rightarrow \Diamond \text{NonFull}) \quad (\text{FullToNonFull}).$$

Model checking this property on the source program directly is relatively expensive due to the state explosion. A popular way to attack this problem is predicate abstraction. This abstraction method observes a program's behavior on a selected set of predicates and translates the statements of the program into statements that operates exclusively on the boolean variables representing the predicates. Bandera [4], for example, can be applied to verify this property. The problem is, can we reuse the verification work done by a tool like Bandera such that when a client download this piece of code, it does not have to run Bandera again?

The right side of Fig. 2 listed the Boolean program of the first loop body, based on the following predicates.

$$\begin{aligned} &\{\text{Empty}_i : \text{buf}_i = 0, \quad \text{where } i = 0, 1. \\ &\quad \text{Full}_i : \text{buf}_i = \text{bound}, \\ &\quad \text{NonFull}_i : \text{buf}_i < \text{bound}\} \end{aligned}$$

2.2 Example 2: Multiple Properties

In reality, many things can go wrong with a program. For example, a mobile program may use a lock; it may read from and write to a file; and it may

```

// int buf1=bound, buf2=0;
// int bound= 50;

thread1 {
  int buf1 = bound;
  run() {
    while (true){
      synchronized {
        while (buf1 <= 0)
          wait();
        buf1--;};

        synchronized {
          while (buf2 >= bound)
            wait();
          buf2++;}
      }
    }
  }
}
// thread2 symmetric to thread1

while 1
  synchronized{
    while (*) {assume(empty1);wait};
    assume (! empty1);
    [
      full1 := H(0,full1 || nonfull1),
      nonfull1 := H(full1 || nonfull1, 0),
      empty1 := *;
    ]
  }
while 1
  synchronized{
    while (*) { assume (full2); wait};
    assume (! full2);
    [
      empty2 := H(0,empty2 || nonfull2),
      nonfull2 := H(0, full2 ),
      full2 := *;
    ]
  }

```

Fig. 2. Bounded Buffer Example and Its Predicate Abstraction (Some omissions are made)

contain a few threads, as the previous example does. For a client to have higher confidence of this program, a server may certify all these properties.

Another scenario: different client may have different needs; they may enforce different security policies. A simple example is that a client may not allow network connection after the mobile program has read from the local disk while this operation is totally legitimate and necessary for another client. Typically, these operations are granted at run time, which certainly will incur some costs. Certification of relevant temporal properties will allow a client to grant or refuse certain operations at link time and save the client from these costs. The problem is: a server may not be able to tell beforehand which client wants which property.

When multiple properties are to be certified, the size of a certificate may become even more significant an issue. Using current techniques, the total size of the certificate will be too large to be transmitted over the internet. In fact, this kind of problem is common for the proof-based certification techniques. Because a proof for a temporal property must deal with every possible path a system may go through, and for software, the number of such paths can be large, the size of the proof is almost inevitably big, even when the property is tested on an abstraction of the program.

For example, BLAST generates small size certificates for reachability programs. But for a single property specified on a windows driver, the size of the program can be as large as more than 100 Mega bytes. BLAST is representative in this aspect of other proof-based approaches.

3 Abstraction as Certificate

Based on the discussion above, we seek other methods to certify a temporal property. One of the obvious choices is to use an abstract interpretation of the mobile program as certificate. In theory, any kind of abstraction that enables successful model checking can serve as the certification method. However, we are going to choose Boolean abstraction. This choice has the following direct benefits.

- A large set of temporal properties can be certified.
- The size of a certificate is controlled. The size of a Boolean abstraction is typically small, linear in the size of the program.
- It is feasible to implement such a system. Abstraction is computed as part of the verification result. Boolean abstraction have been successful in verifying temporal properties on industrial strength programs.
- Verification time can be reasonable. Model checking an abstraction is typically much faster than model checking the source program, if the latter is ever possible. Thus a client may not need very long time verifying the temporal property.

More interestingly, when two sets of predicates are generated for two properties, we can generate a Boolean program using the union of these two predicate sets. A client, however, can be informed of which predicates are for which property. The client can extract a Boolean program from the sum Boolean program and model check the property on it.

ACC represents a local maximal point in the design space. Comparing to proof-based approach, which generates relative larger certificates that are quicker to verify, ACC generates smaller certificates that take longer to verify. More specifically, model checking is needed by ACC to verify the temporal property, while the proof based approach simply verifies the attached proof. An important aspect in evaluating the practicality of ACC is to demonstrate that verification time is acceptable, at least for certain applications.

Another dimension of comparison is the flexibility. With proof-based approach, one proof is generated for one property; the server must know the exact property a client expects to provide a proof. To provide a choice for the client between properties ϕ_1 and ϕ_2 , a server has to provide a proof of $\phi_1 \wedge \phi_2$. With ACC it is possible to provide an abstraction that may certify multiple number of properties, including ones that are not known to the server at all. Detailed discussion on this issue is in Sec 5.

Next we present the ACC framework, with brief introduction to how we have implemented this system.

3.1 Control Flow Automata

We use a control flow automaton (CFA) to model a program's control flow. Intuitively, a CFA comprises: (1) variables, such as local and global variables in

a source program, or memory locations, including registers, in an intermediate program, (2) vertices: control locations of a program, and (3) labeled directed edges that connect vertices. An edge is labeled either with a sequence of instructions that are executed to move from the source to destination locations of the edge, or with a predicate corresponding to the branch condition that must be true for that edge to be taken, or with a symbol for surrendering the CPU in a concurrent setting.

Formally, a CFA is a tuple $(Q, Q_0, X, \text{Op}, \rightarrow)$, where Q is a finite set of control locations, Q_0 is the set of initial control locations, X is a finite set of variables, Op is a set of operations, and $\rightarrow \subseteq (Q \times \text{Op} \times Q)$ is a finite set of edges labeled with operations. The operation allowed is one of the following: (1) a sequence that consists of one or more instructions or statements, (2) a predicate that enables the transition, or (3) a symbol i that indicates the CPU may be surrendered to another thread (this case is only useful in languages with a notion of interleaved concurrency).

A program written in C can be translated into a CFA [14]. Because a Boolean program shares the same CFA as the original program, if we want to associate the intermediate program with the same Boolean program, it is essential to preserve the CFA structure when compiling the source program into an intermediate program.

3.2 From CFA to Transition System

Let $\mathbf{v}_X$ be valuations over X that contain type-preserving functions from X to values. A state is a pair in $Q \times \mathbf{v}_X$. A region is a set of states. Let R range over regions. The location component q in a state (q, V_X) can be viewed as part of X (technically X may need to be expanded). Thus a state is an element in $\mathbf{v}_X$. A transition system induced by a CFA $(Q, Q_0, X, \text{Op}, \rightarrow)$ will be a tuple $(\mathfrak{R}, R_0, \rightarrow)$, where $\mathfrak{R}$ here represents the set of all possible states, R_0 is the set of the initial states, and $\rightarrow$ is reused to represent the transition relation defined by edges $\rightarrow$. Note that we may refer to this transition relation as the post operator. Accepting states are not defined in this system.

Standard techniques allow us to determine if this transition system satisfies an LTL formula ϕ , provided that we can evaluate the atomic propositions of ϕ at a given state. We first construct a Büchi automaton that accepts the negation of ϕ . Then we construct a product automaton of the Büchi automaton and the transition system. Model checking this automaton will evaluate ϕ .

So far we have not distinguished a concrete program and an abstract program. An abstract program is induced by an abstraction of the state space X . The abstraction, as usual, is defined by a pair (α, γ) that forms a Galois connection between all concrete states and all abstract states. The abstraction function α will map a concrete region to an abstract one. The pair induces an abstract transition relation $\rightarrow_a$ between abstract regions. We thus can compute an abstract transition system defined by an abstract state space and $\rightarrow_a$. We can model check the abstract transition system to verify the property on the concrete system. Because of the Galois connection requirement, the relation $\rightarrow_a$ is

an over approximation of $\rightarrow$. As a result, if we model check the abstract transition system, only global properties, such as LTL formulas, can be verified on the original system.

ACCEPT adopts Boolean abstraction, where the abstract transition $\rightarrow_a$ is defined by a Boolean program[2]. Usually, for a software system, most abstractions will verify a next operator-free LTL formula.

3.3 Concurrency

Assume a source statement is atomic, that is, modifying at most one shared variable¹. We can apply ACC to concurrent programs. The Boolean program translation of a statement will correspond to an atomic block in a language such as Promela.

For every statement in a parallel thread, we generate an i transition to any other statement in other thread. Such a transition may or may not be enabled during model checking. For example, the model checker may only choose the next statement of a pending thread. Even so, the introduction of this i causes the model checker to consider too many choices and slows down the model checking significantly.

One way to attack this problem is through thread-modular verification, based on assume-guarantee principle [18], as implemented by a few state-of-art systems [9, 6, 7]. Although these systems have been successful, not all temporal properties can be automatically checked this way. Our approach is to analyze the concurrent primitives to reduce the number of i transitions. For example, when two critical sections are protected by the same lock, and we know mutual exclusion holds, we can delete the i transitions from a statement in the critical sections to statements in the other critical section. This idea is partially tested in ACCEPT by certifying liveness properties on some toy concurrent programs..

Another issue is that model checking a concurrent Boolean program (with function call) may not terminate, which, however, is not within the scope of this paper.

3.4 On Implementation of ACC

We have answered a few questions necessary to implement an ACC system. First, a mobile program is often compiled. Directly computing an abstraction of an intermediate program suffers from the fact too many predicates are needed. How do we compute a useful abstraction for an assembly or an intermediate program? Second, Is there a way to represent the certificate, a Boolean program, in an efficiently verifiable way? We will devise an abstraction-preserving compilation scheme with which we can reuse a Boolean program for the source program as a valid abstraction of an assembly program. We will use index types in an index type assembly language to encode the Boolean program. An index type type

¹ A similar assumption is anyway needed for the soundness of abstraction based model checking.

checker is invoked to validate the encoded Boolean program. These have been covered by our sister papers [23, 22].

We implemented ACC system in ACCEPT. We present our experience below.

4 Experience

We have constructed two instances of the ACCEPT framework. The first version, ACCEPT/While, implements a simple compiler for a while language with concurrency primitives. It generates programs for an index-typed JVM dialect that we developed earlier. The second version, ACCEPT/C, leverages the BLAST infrastructure, including CIL. It operates on C language programs. It also produces index-typed JVM code.

The basic components of ACCEPT include an encoder that encodes Boolean programs into type annotations is implemented together with the type checker of SDTAL.

Initial results with ACCEPT/While are reported in Table 3, which lists the size of the certificate generated for several well studied concurrent programs and some liveness properties. The percentage indicates that, even when the program does not contain much irrelevant code (which a realistic program often does), the size of the certificate is small. Also, we listed the verification time spent on model checking Promela programs translated from the abstracted programs. The measurement is done on a Pentium III 700 Hz laptop with 256MB memory running Linux.

	Program Size	Certificate Size	Percentage	Verification Time
Bounded Buffer	702	232	33%	0.42
Vector Addition	808	308	38%	0.30
Reader/Writer	1570	282	18%	0.43
Sleeping Barber	809	200	25%	0.74

Fig. 3. Size of Certificate in Bytes

Recently, we migrated BLAST [10] on top of ACCEPT infrastructure, resulting in ACCEPT/C. We are able to reuse the basic components of ACCEPT. BLAST handles the full syntax of C programs through the use of CIL [14]; BLAST adopts lazy abstraction, a variation of predicate abstraction, to verify reachability properties of industrial strength programs, such as device drivers. BLAST represents C programs as labeled transition systems built from CFAs during model checking. We compute a Boolean program from BLAST's intermediate result, after BLAST has gone through the refinement based predicate discovery process to settle down with a set of predicates with which a useful abstraction is made. Then we compile the CFA representation to JVM code

and encode the Boolean program as index type annotations. On the client side, ACCEPT's SDTAL type checker will validate the Boolean program.

The lazy abstraction is a dynamically (in the process of model checking) scoped predicate abstraction. When we generate Boolean programs, we take the sum of those predicate sets that are used to abstract the same operation; the results are statically scoped predicate sets. We use these predicate sets to abstract the C program.

We are able to generate relatively small certificates. For example, for the correct use of locks, the certificates generated for most Linux drivers, whose sizes range from a few hundreds lines to hundreds of thousands lines, range from hundreds of bytes to a few thousands bytes. The figure is smaller than the size of certificates generated by BLAST. Note that BLAST only certifies source programs. We compare our result with BLAST here assuming that BLAST has adopted our compilation scheme and that the proof it generates is valid for the intermediate program.

Another way to compare with BLAST will count the number of verification conditions that a certificate induces. With BLAST, the number is the size of the search tree because a location invariant in BLAST needs regions from every node in the search tree corresponding to the location. Our certificate is linear in the size of the relevant part of the program. For correct use of locks, a search tree often collapses into a chain because of the typical way a lock is used. For such applications, BLAST is able to generate very small certificates. For more complicated properties, such as the IRP specification of Windows NT drivers, a typical proof generated by SLAM can be about 100 Mega bytes, which can be compressed to about a few Mega bytes. A typical ACCEPT/C certificate is about 100 Kilo bytes before compression.

The price we pay is the time spent on model checking. While BLAST verifies a proof of the size of the search tree, we need to generate the search tree. This means calling a decision procedure as many times as the size of the search tree in the worst case. In practice, such search doesn't take long[10], ranging from less than a second to a few seconds for checking the correct use of locks for Linux device drivers. In Table 4, we list the results of certifying a few Linux device drivers using our system. The verification time is the model checking time reported by BLAST. The numbers can be improved if a quicker model checker is adopted. The program size is the number of lines of code after we pre-process the source program. Certificate size is measured in bytes. Our experience confirmed that the verification time is acceptable for some applications.

Negative results do appear in our experience. For example, the Windows device driver examples often take more than a few hundreds of seconds to model check. When that happens, the choice between ACCEPT/C and BLAST depends on how long it takes to download the Mega-byte proof generated by BLAST. Still, the solution based on ACC may be viable in certain circumstances. For example, the client might tolerate hundred-second model checking time and the network connection is rather slow.

To summarize, using ACCEPT/C, we obtain a practical solution to the certification of reachability properties on realistic C programs. This solution trades model checking time for certificate size.

	Program Size	Our Certificate Size	BLAST Certificate Size	Verification Time
atp.c	2482	1212	8737	1.31sec
ide.c	48k	876	12452	0.37sec
audio.c	175k	15410	502575	5.60sec

Fig. 4. Comparison between ACCEPT/C and BLAST

5 Discussion

Adopting Boolean abstraction as the certification method, we can certify multiple properties by constructing a sum abstraction. A sum abstraction of two Boolean abstraction based on predicate set P_1 and P_2 is a predicate abstraction based on $P_1 \cup P_2$. A Boolean program based on predicate set P can be projected to a Boolean program based on Q , where $Q \subseteq P$. The projection can be recursively defined over expressions as following.

- A boolean variable b is projected to b if $b \in Q$, or $*$ if not.
- Negation, disjunction and conjunction is recursively defined over its operands. When $*$ is involved, $\neg * = *$; $a \wedge * = *$ and $a \vee * = a$.

It is relatively easy to extend this definition over **assume** and assignment statements in a Boolean program. Such a projection operation is expected to extract a Boolean program based on Q from the certificate. This way, we have a method to certify and selectively verify multiple properties. Note that this is however, not unique to ACC. With proof based approach, we can also extract one proof of a property from a proof for conjunctions of multiple properties. The difference is still about the size of the certificate.

Strictly speaking, just as a proof is directly linked to a property, in a counter-example guided refinement predicate discovery system, a resultant Boolean program is directly linked to a temporal property. The concept of ACC, in general, allows us to separate a server from knowing the exact policy of a client because multiple properties can be verified on one abstraction. To fully realize this potential, we need research on feed-back free predicate discovery, which we conjecture might be possible for some properties.

Another possible future work is to reduce the size of a Boolean program. In a system like SLAM or BLAST, refinement-based predicate discovery is performed to find a right set of predicates. A new predicate is introduced by applying a

so-called focus operator to predicates. When this process is performed automatically, many predicates are in fact related. Binate covering is a technique commonly used in digital circuit optimization. Given a boolean function, binate covering finds a minimum representation of this function, subject to a set of constraints. In our setting, the constraints are the relation among predicates. For example, Full and NonFull in our early example are related, Full implies the opposite of NonFull. Finding a minimum representation of the post operator minimizes the size of the certificate, which in turn reduces the cost of model checking.

6 Conclusion

In this paper, we describe abstraction-carrying code, which offers a method that may generate small size of certificates for temporal properties of sequential and concurrent programs. We summarize some technical development involved in designing and implementing an ACC system. Some experiences of application of ACC are also reported. We suggest the application of ACC to the certification of multiple temporal properties, which can lead to an separation of knowledge between a client and a server. We discuss the technical issues related to this new possibility, as well as other techniques that may improve the performance of an ACC system.

7 Acknowledgement

A long list of people offered help to our research. The idea originated in a class taught by Mike Gordon two years ago and was later sharpened by discussion with Penix, Wesser, Ball, Amy Felty, Rajamani, Andrew Bernard and Hatcliff. Hatcliff also sent us a simple implementation of predicate abstraction to help us understand the technique. Special thanks to Rupak Majumdar, who provided newest version of BLAST.

Discussions with our colleagues at Pacsoft, especially with John Matthews and Mark Jones, are both informative and encouraging. Thank Tim Sheard for his tutoring the first author on English writing.

References

1. A. Appel. Foundational Proof-carrying Code. In *Proceeding of 16th IEEE Symposium on Logics in Computer Science*, June 2001.
2. T. Ball and S. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *SPIN2001, Lecture Notes in Computer Science, LNCS2057*, pages 103–122. Springer-Verlag, May 2001.
3. A. Bernard and P. Lee. Temporal Logic for Proof-carrying Code. In *Proceedings of the 18th International Conference on Automated Deduction (CADE-18)*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 31–46, Copenhagen, Danmark, July 2002.

4. J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Extract Finite-state Models from Java Source Code. In *Proceedings of ICSE 2000*, 2000.
5. S. Das, D. Dill, and S. J. Park. Experience with Predicate Abstraction. In *Proceedings of CAV'99, LNCS 1633*, pages 160–171, Trento, Italy, July 1999.
6. C. Flanagan, S. Freund, and S. Qadeer. Thread modular verification for shared memory programs. In *Proceedings of ESOP 02*, 2002.
7. C. Flanagan and S. Qadeer. Thread-modular model checking. In *Proceedings of SPIN 2003*, 2003.
8. S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Proceedings of CAV'97, Lecture Notes in Computer Science, LNCS 1254*, pages 72–83, Haifa, Israel, June 1997. Springer-Verlag.
9. T. Henzinger, R. Jhala, R. Majumdar, , and S. Qadeer. Thread-modular abstraction refinement. In *Proceedings of CAV 03*, 2003.
10. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, pages 58–70, 2002.
11. Kedar S. Namjoshi. Certifying Model Checkers. In *13th Conference on Computer Aided Verification (CAV)*, 2001.
12. Kedar S. Namjoshi. Lifting Temporal Proofs through Abstractions. In *VMCAI*, 2003.
13. D. Kozen. Efficient Code Certification, 1998. Technical Report, Computer Science Department, Cornell University.
14. S. McPeak, G. C. Necula, S. P. Rahul, and W. Weimer. Cil: Intermediate languages and tools for c program analysis and transformation. In *Proceedings of Conference on Compiler Construction (CC'02)*, March 2002.
15. G. Morrisett, K. Crary, N. Glew, and D. Walker. Stacked-based Typed Assembly Language. In *Proceedings of workshop on Types in Compilation, also in Lecture Notes in Computer Science, LNCS 1473*, pages 28–52. Springer Verlag, 1998.
16. G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.
17. G. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, 1998.
18. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, pages 319–340, 1976.
19. Z. Shao, B. Saha, V. Trifonov, and N. Papaspyrou. Type System for Certified Binaries. In *Proc. 29th ACM Symposium on Principles of Programming Languages (POPL'02)*, pages 217–232, January 2002.
20. L. Tan and R. Cleaveland. Evidence-based model checking. In *13th Conference on Computer Aided Verification (CAV)*, pages 455–470, 2001.
21. H. Xi and R. Harper. Dependently Typed Assembly Language. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, pages 169–180, Florence, September 2001.
22. S. Xia and J. Hook. Experience with abstraction-carrying code. In *Proceedings of Software Model Checking Workshop*, 2003.
23. S. Xia and J. Hook. An implementation of abstraction-carrying code. In *Proceedings of Foundations of Computer Security Workshop*, 2003.

Instrumentation of Java Bytecode for Runtime Analysis

Allen Goldberg and Klaus Havelund

Kestrel Technology, NASA Ames Research Center
Moffett Field, MS 269-3, California USA

Phone: 650-604-4858, Email: {goldberg,havelund}@email.arc.nasa.gov

Abstract. This paper describes **JSpy**, a system for high-level instrumentation of Java bytecode and its use with **JPaX**, our system for runtime analysis of Java programs. **JPaX** monitors the execution of temporal logic formulas and performs predicative analysis of deadlocks and data races. **JSpy**'s input is an instrumentation specification, which consists of a collection of rules, where a rule is a predicate/action pair. The predicate is a conjunction of syntactic constraints on a Java statement, and the action is a description of logging information to be inserted in the bytecode corresponding to the statement. **JSpy** is built using JTrek an instrumentation package at a lower level of abstraction.

1 Introduction

This paper describes an instrumentation package, **JSpy**, for instrumenting Java [3] bytecode [11], and our application of **JSpy** to *run time analysis*. Run time analysis [1,2] refers to examination of a single program execution trace to check, for example, for conformance to a temporal logic assertions, or for detecting the potential for deadlocks and data races. The Java PathExplorer, **JPaX** [10], is our general framework for performing runtime analysis. **JPaX** has been incorporated into an automatic test case generation system. See [4] for details.

JSpy can be seen as an Aspect Oriented Programming [9] environment package in the sense that it is guided by rules, or *aspects*, which specify how a program should be transformed to achieve additional functionality. However, the main purpose of these aspects is to extract information from a running program. **JSpy** is built on top of the low-level JTrek instrumentation package [7]. Other low-level instrumentation packages could be used, such as for example BCEL [8].

The paper is organized as follows. Section 2 gives a description of the overall functionality of **JSpy**, and of the input specification that drives it. Section 3 discusses some implementation issues and how to run it. Section 4 lays out an architecture for using **JSpy** to generate events to monitors, that check for temporal logic property conformance and multi-threading problems. Section 6 discusses some planned enhancements and gives conclusions.

2 Input Instrumentation Specification

JSpy produces instrumented code based on an *instrumentation specification* whose main component is a set of rules. We refer to the code to be instrumented as the *target* and the result of instrumenting the target the *instrumented target*. The instrumented target is functionally equivalent to the target except that, when executed it generates a stream of *event log objects* as specified by the rules of the instrumentation specification. The rules form a high-level declarative description of where to insert log creation code and the contents of the log entries. A design goal for the instrumentation package is to minimize the impact of the instrumentation code on the time and space of the target program. Our run time analysis system is designed so that the substantial work of analyzing event logs is done off-line via file communication, or by a remote process that reads the log stream via a socket connection.

Each rule is a predicate/action pair. The predicate is a conjunction of atomic predicates. Each atomic predicate defines a syntactic constraint on the sequence of bytecode instructions corresponding to a single Java statement. Examples of predicates are summarized in Figure 1. A name specification, for example *methodNameSpec*, specifies a fully qualified method name and method signature but allows wildcards for most components. Actions specify instrumenting directives that are inserted at statements satisfying the rule predicate. Example actions are given in Figure 2. The action *ReportExpression* takes an arbitrary Java expression, encapsulates it into a method and inserts a call to the method into the target. The method is inserted into a generated Java file inserted into the package containing the target code. By placing the code in the same target as the target class file most data values available to the target code will be visible to such a method. Other data, e.g. private data or local variables can be transmitted as a parameters to the constructed method. When this auxiliary class is completely populated with such methods, the Java compiler is called and the resulting bytecode becomes part of the instrumented target.

An example instrumentation specification, using the **JSpy** Java API, is illustrated in Figure 3. An instrumentation specification is built up from one or more rules, each being a set of predicates and a set of actions that are inserted if the predicates all evaluate to true. The example specification contains one rule, with one predicate and two actions.

The goal of minimizing the impact of instrumentation on execution time affects a fundamental design decision regarding the logging of program data in response to a *ReportLocal*, *ReportField* or *ReportExpression* action. If the type of value to log is primitive then the value can be inserted into the log. Since the size of the value is just a few bytes the log entry is itself small. For values that are objects there are four choices:

- A deep copy of the object is transmitted. That is, all fields of the object are transmitted, including all objects reachable from the fields of the object.
- A shallow copy is transmitted. That is, the values of primitive-type fields and an external reference to the values of non-primitive fields are transmitted.

Predicate Name	Parameters	Returns <i>true</i> iff. current statement:
InClass	String <i>classNameSpec</i>	is within class with name that matches <i>classNameSpec</i> .
InMethod	String <i>methodNameSpec</i>	is within method with name that matches <i>methodNameSpec</i> .
AtMethodStart	String <i>methodNameSpec</i>	is the first statement within method with name that matches <i>methodNameSpec</i> .
AtMethodEnd	String <i>methodNameSpec</i>	is the last statement or a return statement within method with name that matches <i>methodNameSpec</i> .
CallsMethod	String <i>methodNameSpec</i>	contains a call to a method who's name matches <i>methodNameSpec</i> .
AtFieldAccess	String <i>fieldNameSpec</i> boolean <i>onlyUpdates</i>	accesses a field who's name matches <i>fieldNameSpec</i> . If <i>onlyUpdates</i> is true, only write accesses are matched.
AtLocalAccess	String <i>varNameSpec</i> boolean <i>onlyUpdates</i>	accesses a local variable who's name matches <i>varNameSpec</i> . If <i>onlyUpdates</i> is true, only write accesses are matched.
AtSyncStart		enters a synchronized block, taking a lock.
AtSyncEnd		exits a synchronized block, releasing a lock.
AtStatementType	String <i>stmtType</i>	is a statement of type <i>stmtType</i> , which ranges over 32 different statement types, such as: "assign", "break", "return", "try", "throw", "while", etc.
InStatementRange	int <i>lowerBound</i> int <i>upperBound</i>	is within the range of line numbers <i>lowerBound</i> and <i>upperBound</i> .

Fig. 1. Examples of Predicates

- An external reference to the object is transmitted.
- Object values are not transmitted at all.

For some applications transmitting deep copies may be appropriate, e.g. for off-line program debuggers. Our approach is to just transmit an external reference. Note, using *ReportExpression*, the value of any field is transmittable, so the effect of a shallow copy is achievable. Not transmitting object values is too limiting. In particular, it is necessary to log object references in order to properly log the taking and releasing of locks on those objects, information that is needed for example for deadlock and data races analysis.

However this raises a problem of how to externally reference an object. Java serialization is a potential solution to this since it provides external unique identifiers to objects, but serialization externalizes a portion of the heap at exactly one time instant. For our purposes a reference to the same object reported at

Action Name	Parameters	Inserts code that reports:
ReportMethodStart	String <i>methodNameSpec</i>	name of method entered, who's name matches <i>methodNameSpec</i> .
ReportMethodEnd	String <i>methodNameSpec</i>	name, and return value, of method exited, who's name matches <i>methodNameSpec</i> .
ReportMethodCall	String <i>methodNameSpec</i>	name of method that is called, who's name matches <i>methodNameSpec</i> .
ReportField	String <i>fieldNameSpec</i> boolean <i>onlyUpdates</i>	name of field that is accessed, who's name matches <i>fieldNameSpec</i> . If <i>onlyUpdates</i> is true, only write accesses are matched.
ReportLocal	String <i>varNameSpec</i> boolean <i>onlyUpdates</i>	name of local variable that is accessed, who's name matches <i>varNameSpec</i> . If <i>onlyUpdates</i> is true, only write accesses are matched.
ReportSyncStart	String <i>classNameSpec</i>	identity and class (name) of object that is locked, where the class name matches <i>classNameSpec</i> .
ReportSyncEnd	String <i>classNameSpec</i>	identity and class (name) of object that is released, where the class name matches <i>classNameSpec</i> .
ReportTimeStamp		the current time.
ReportProgramPoint		the current line number.
ReportExpression	String <i>expressionName</i> String <i>expressionBody</i> byte <i>expressionType</i> String <i>parameters</i>	the value of <i>expressionBody</i> , and an identifier <i>expressionName</i> . <i>parameters</i> and <i>expressionType</i> indicate the expression's free parameters and the result type.

Fig. 2. Examples of Actions

two points during execution must be the same, and this is not within the scope of serialization.

Unfortunately we do not have a completely satisfactory solution for constructing such an external reference. Our approach is to transmit as an external reference the hash code of the object, when viewed as an instance of `java.lang.Object`. This value is computed by the method `java.lang.System.identityHashCode(Object o)`. There is the remote chance that two distinct objects are assigned the same hash code. We decided that this imperfect solution is preferred to disallowing the transmission of object values. We treat strings as a special case, transmitting the whole object when requested.

An additional feature of **JSpy** is that actions can be conditionally executed. A conditional action is specified by providing a boolean-valued Java expression that gets compiled using the same mechanism as described for *ReportExpression*. The target is instrumented so that a call is made to a method at execution time, whose body uses the boolean expression to guard the logging action. In some

```

class LogLockAcquisitions{
    ...

    public void instrument(){
        InstrumentSpecification spec = new InstrumentSpecification();
        Rule rule = new Rule();
        rule.addPredicate(new AtSyncStart());
        rule.addAction(new ReportTimeStamp());
        rule.addAction(new ReportSyncStart("*"));
        spec.addRule(rule);
        Instrumenter ins = new Instrumenter(spec);
        ins.instrumentTheCode();
    }
}

```

Fig. 3. Example Instrumentation Specification

simple cases encapsulation can be avoided in favor of a low-overhead conditional placed directly in the code.

Some actions do not make sense in certain contexts. For example the action *ReportMethodStart* only makes sense instrumenting the first statement of a method. Consequently such actions will only be inserted at appropriate program points.

3 Implementation

In this section we discuss some implementation issues and how an instrumented program is run.

3.1 JTrek

JSpy is built using Jtrek [7]. Jtrek is a lower level bytecode modification tool with monitoring being one of the intended applications. JTrek was developed at COMPAQ SRC. Jtrek iterates through the bytecode instructions of the target and uses callbacks to perform user-specific instrumentation. JTrek allows insertion of certain types of code, but does not allow definition of new local variables, fields, or methods. Since calls to arbitrary methods are supported, it is possible to encapsulate any instrumentation action into a method that is inserted into the code.

Iteration may be at the level of Java statements or individual bytecode instructions. Jtrek limitations have made it difficult or impossible to cleanly implement some of the features of **JSpy**, but overall, it is a stable and well-designed software package.

As described above Jtrek provides an idiom of use based on an iteration, or trek, over the bytecode. **JSpy** is structured as such a trek; for each bytecode

translation of a Java statement the predicate of each rule of the instrumentation specification is tested. The actions of all rules whose predicates are true are collected, sorted according to a canonical ordering on actions, and inserted into the code using Jtrek primitives. In most cases the inserted code is a call to a method that generates an entry into the event stream. For example, consider the implementation of the *ReportLocal* action for a field of type boolean. The code inserted will invoke a static method called *reportLocalBoolean* with three parameters, a string holding the name of the class and method in which the variable is defined, a string containing the local variable name, and the (boolean) value itself. This method will write a log entry.

Jtrek iterates over multiple class files; specifying the target classes to be instrumented is achieved in a manner similar to Java class loading. A root class, classpath, and a scope indicator are given. The scope indicator can be “all”, “user”, or “package”. All files directly and indirectly referenced in the root class and consistent with the scope designator are instrumented. The “package” designator restricts instrumentation to files in the same package as the root class, “user” excludes instrumentation of system files, “all” means all reachable files are instrumented.

3.2 Execution of the Instrumented Target

As described above, the target is instrumented with calls to methods that construct a log entry and then write the entry to an output stream. In addition, auxiliary classes are constructed “on the fly” as necessary, and added to user packages containing the target code. Finally the instrumented target contains predefined supporting classes where such methods as *reportLocalBoolean* are found. Within the supporting classes is an abstract class of log objects and specializations of this class for each type of action.

The instrumented target is started as usual, e.g. an application is started by invoking the “main” method of the specified class. When an instrumentation method is first encountered in the code, the instrumentation class is loaded and a static initializer for the class is executed. This code attempts to read a property file that describes where the log output stream is routed. If no such file exists, then default routing is used. The output stream is routed to either a socket or file. The output stream consists of serialized log objects. The static initializer also sets up a shutdown hook (see `java.lang.Runtime.addShutdownHook`) to flush and close the buffered output stream when the application completes.

Synchronization occurs at the finest level of granularity, namely writing the log entry to the buffered output stream. This minimizes the effect of synchronization on performance of multi-threaded targets. However, as a consequence, events from different threads may be interleaved. To enable re-creation of events at the receiving end, each log entry contains an external reference to the executing thread.

3.3 Logging Monitor Exit

One of our applications of **JSpy** is run time analysis of programs for deadlocks and data races. This requires logging the taking and releasing of synchronization locks. At the JVM level, locks are obtained when a synchronized method or `monitorenter` instruction is executed. As stated in the JVM specification [11]: “Normally, a compiler for the Java programming language ensures that the lock operation implemented by a `monitorenter` instruction executed prior to the execution of the body of the synchronized statement is matched by an unlock operation implemented by a `monitorexit` instruction whenever the synchronized statement completes, whether completion is normal or abrupt.” Thus instrumenting all `monitorenter` and `monitorexit` correctly tracks the number of locks held by a thread on an object relating to `synchronized` statements. However if a synchronized method abruptly terminates, then the lock obtained on entry to the method is released by the JVM, but there is no way to instrument the bytecode to record the release of the lock. The not-to-elegant fix to this problem is to modify the body of each synchronized method to surround the whole body with as `try` block that catches all throwables, logs the release of the lock, and rethrows the exception.

4 Writing Monitors

The Java PathExplorer, **JPaX**, in which **JSpy** has been applied, provides a general framework for writing program execution monitors. One kind of monitoring is conformance checking against temporal logic formulas. Temporal logic formulas are assertions about a sequence of states, intended to describe properties of programs [12]. Formulas of temporal logic are constructed from atomic formulas using Boolean operators and temporal operators such as $\diamond$ (meaning eventually in some future state). In propositional temporal logic the atomic formulas are predicates on states. Using techniques such as model checking, a program may be verified with respect to a temporal logic formula, by showing the all possible execution traces of the program satisfy the formula. Run time analysis checks that one execution of the program satisfies the formula. This section examines practical considerations of mapping the meaning of satisfaction in temporal logic to Java programs. For propositional temporal logic this concerns how to define sequences of states and predicates over such states. The “obvious” correspondence is that a state is the computation state of the JVM, i.e. the state of the heap, invocation stacks and instruction counters for threads, etc. Transition to a new state is achieved by execution of a JVM instruction. Immediately one practical problem is clear: state transition at the level of individual instructions is too fine grained. If at each instruction cycle of the JVM a log entry is transmitted then the instrumented target will execute many times slower than the un-instrumented target.

Our objective of limiting the impact of instrumentation on program execution time leads to our design decision to check satisfaction of temporal formula off-line. We call the off-line checker an *observer*. The design of **JPaX** includes a

dispatcher which routes the event stream to one or more such observers. This architecture implies that atomic predicates are restricted to the information about the JVM state obtainable from the **JSpy** log stream. In fact, it is desirable to define within each observer a data structure which records an *observer state* that is updated by the log stream. Thus, the observer state is an abstraction of the state of the JVM. The predicates of the propositional formula are viewed as predicates over the observer state, rather than the JVM state.

Propositional temporal logic has expressibility limitations that impact its practical application. For example, it is impossible with propositional temporal logic to count the number of times a predicate is true within a computation. However one can add components to the observer state that perform such counting and reference these components in the definition of propositional predicates. Thus a temporal logic observer is defined by defining

- an instrumentation specification,
- an observer state,
- how the observer state is updated by the event stream,
- predicates over the observer state,
- a temporal logic formula over the predicates

We have defined a simple extension to propositional temporal logic that introduces a binding operator. The binding operator binds a variable in a formula to a component of the observer state. Predicates within the scope of the binding operator may be parameterized by the variable.

JPaX furthermore includes an observer that can detect the potential of a program to contain a deadlock or data race. Such an observer requires an event stream that includes taking and release of locks and updates to shared objects. The state of the observer consists of data structures such as lock graphs. Details can be found in [5,6,10].

5 Comparison with Other Approaches

The intended application for **JSpy** is **JPaX** and so our primary concern designing **JSpy** was insuring that application was well supported. Logging is a classic application of aspect-oriented programming, but AOP systems like AspectJ do not have the needed flexibility for defining code insertion points required by **JPaX**. Furthermore AspectJ modifies Java source, rather than bytecode a significant usability concern, and one that inhibits deadlock and data race analysis. Java debugging and profiling interfaces are either too slow or too restricted. Modification of the JVM is another possible approach, but that is a significant undertaking and one that prevents adoption barriers.

6 Conclusion

We presented an instrumentation package and showed how it is used within our runtime analysis system **JPaX**. We are planning to investigate minimizing the

impact of instrumentation on the execution time of a program, by instrumenting different parts of the code for different test executions and then correlating the results. We also plan to combine dynamic analysis with static analysis.

References

1. 1st CAV Workshop on Runtime Verification (RV'01). In K. Havelund and G. Roşu, editors, *Proceedings of Runtime Verification (RV'01)*, volume 55(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.
2. 2nd CAV Workshop on Runtime Verification (RV'02). In K. Havelund and G. Roşu, editors, *Proceedings of Runtime Verification (RV'02)*, volume 70(4) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2002.
3. K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.
4. C. Artho, D. Drusinsky, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, G. Roşu, and W. Visser. Experiments with Test Case Generation and Runtime Analysis. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003, LNCS 2589, Taormina, Italy*, pages 87–107. Springer, March 2003.
5. C. Artho, K. Havelund, and A. Biere. High-Level Data Races. In *VVEIS'03, The First International Workshop on Verification and Validation of Enterprise Information Systems*, April 2003. Angers, France.
6. S. Bensalem and K. Havelund. Reducing False Positives in Runtime Analysis of Deadlocks. Submitted for publication, October 2002.
7. S. Cohen. Jtrek. Developed by Compaq.
8. M. Dahm. BCEL. Compaq, <http://jakarta.apache.org/bcel>.
9. T. Elrad, R. E. Filman, and A. Bader. Aspect-Oriented Programming. *Comm. ACM*, 44(10):29–32, 2001.
10. K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of the First International Workshop on Runtime Verification (RV'01)*, volume 55(2) of *Electronic Notes in Theoretical Computer Science*, pages 97–114, Paris, France, July 2001. Elsevier Science.
11. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1999.
12. A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.